Stratified Datalog and Program Analysis

Anders Schlichtkrull, René Rydhof Hansen, Flemming Nielson

Abstract

In this entry we formalize stratified Datalog, the first such formalization in Isabelle to the best of our knowledge. Next we formally establish the existence of least solutions for any stratified Datalog program, essential for reasoning about negations in clauses. Lastly we illustrate the usefulness of our Datalog formalization by further formalizing the general Bit-Vector Framework and formalize and prove correct five analyses in this framework namely liveness, reaching definitions, available expressions, very busy expressions and reachability. Many of our definitions follow Nielson and Nielson's textbook [NN20]. The formalization is described in our SAC 2024 paper [SHN24].

Contents

1 Introduction

In the following we develop the first, to the best of our knowledge, Isabelle formalization of Datalog in general and stratified Datalog in particular. We use the formalization to prove the existence of least solutions (to Datalog programs) with respect to a simple ordering on predicate valuations. The ordering is from Nielson and Nielson's textbook [NNH99], and appears also in a paper by Przymusinski [Prz88] in a different formulation. We prove the two formulations equivalent. Furthermore, we apply our formalization to also formalize so-called *bit-vector analyses* [NNH99] as Datalog programs covering all the usual variants [NN20]: forward may analyses, backward may analyses, forward must analyses, and backward must analyses. The analyses are proved to correctly summarize the paths in the program, and we additionally show that each variant of our Bit-Vector Framework has an instance by formalizing four classic analyses: reaching definitions, live variables, available expressions, and very busy expressions [NNH99]. We use the labeled transition systems formalization by Schlichtkrull et al. [SSST23, SSST22] to represent program graphs.

A Rocq formalization of stratified Datalog, based on the thesis by Dumbrava [Dum16], has been developed by Benzaken et al. [BCD17], giving a comprehensive formalization focusing on correctness of evaluation strategies for (stratified) Datalog programs.

Non-stratified Datalog has been independently formalized in Rocq for use as a stepping stone for formalizing further logics (Binder, Horn clauses, BCiC) used to model and reason about authentication [Whi06] and a subset called Regular Datalog has also been formalized in Rocq [BDA18]. Rocq has also been used to formalize various semantics for Prolog (with corresponding proofs of equivalence), with the goal of proving correctness of static analyses of Prolog programs [KKB13]. Non-stratified Datalog has also been formalized in Lean with the goal of certifying results from Datalog reasoners [TGMK25].

Formalizations of analysis frameworks and the underlying theory have mainly focused on the *abstract interpretation* [CC77] approach to static analysis and primarily using Rocq, with impressive results [CP10, JLB+15]. Additionally an Isabelle formalization of abstract interpretation of a small imperative language occurs in the "fully mechanized" semantics textbook of Nipkow [NK14, Nip12].

Concrete program analyses have been formalized and proven correct in Isabelle, as witnessed in the Archive of Formal Proofs[1], including slicing, control flow analysis, conflict analysis, call arity analysis etc. [LMO07, Was08, Bre16, Was09, Bre10, LM08, Imm11, Bre15, Jia21, WL08, WLS09]. These do not use Datalog.

[1] https://www.isa-afp.org/topics/computer-science/programming-languages/static-analysis/

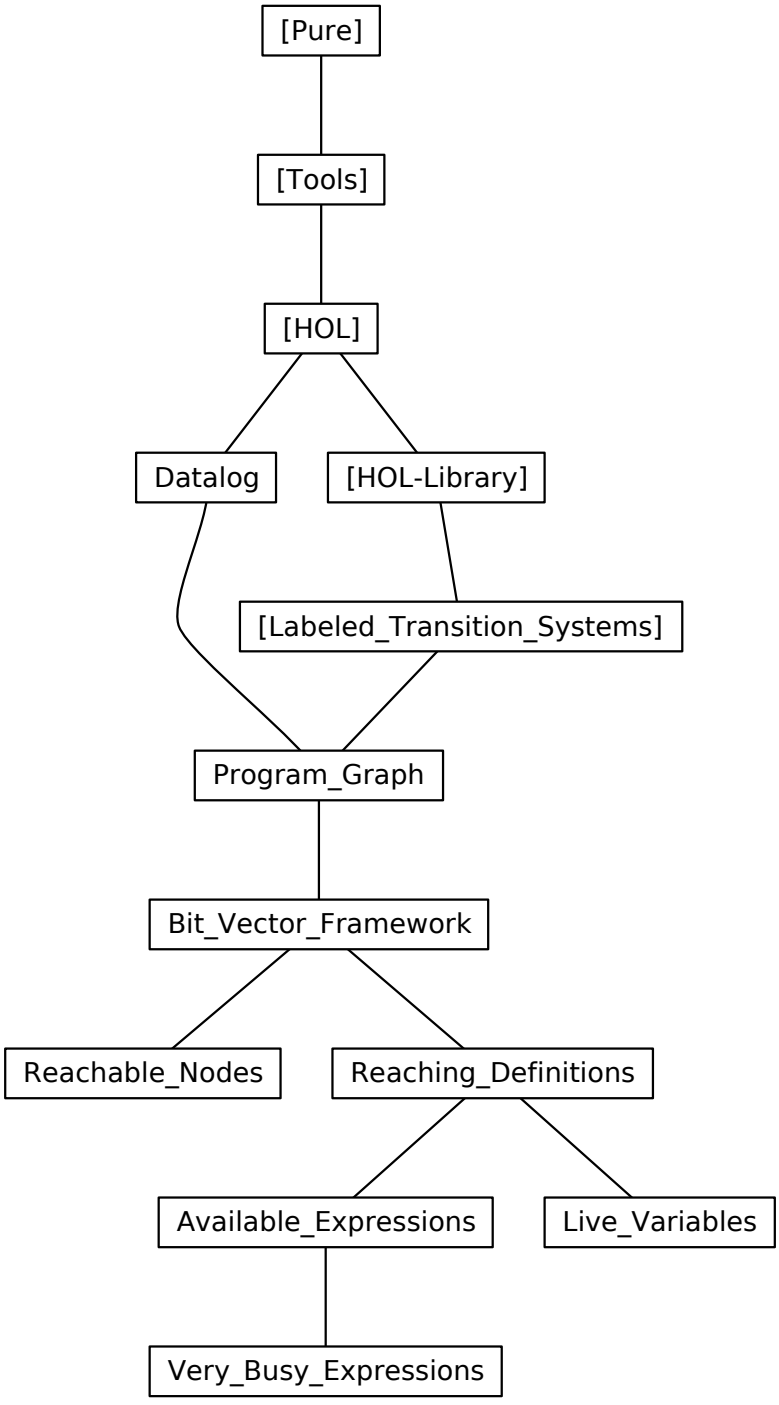

Figure 1: Theory dependency graph

theory Datalog **imports** Main **begin**

2 Datalog programs and their solutions

```
datatype (vars_id: 'x,'c) id =
  is_Var: Var 'x
  | is_Cst: Cst (the_Cst: 'c)

datatype (preds_rh: 'p,'x,'c) rh =
  Eql "('x,'c) id" "('x,'c) id" ("_ = _" [61, 61] 61)
  | Neql "('x,'c) id" "('x,'c) id" ("_ ≠ _" [61, 61] 61)
  | PosLit 'p "('x,'c) id list" ("+ _ _" [61, 61] 61)
  | NegLit 'p "('x,'c) id list" ("¬ _ _" [61, 61] 61)

type_synonym ('p,'x,'c) lh = "'p × ('x,'c) id list"

fun preds_lh :: "('p,'x,'c) lh ⇒ 'p set" where
  "preds_lh (p,ids) = {p}"

datatype (preds_cls: 'p, 'x,'c) clause = Cls 'p "('x,'c) id list" (the_rhs: "('p,'x,'c) rh list")

fun the_lh where
  "the_lh (Cls p ids rhs) = (p,ids)"

type_synonym ('p,'x,'c) dl_program = "('p,'x,'c) clause set"

definition "preds_dl dl = ⋃{preds_cls c| c. c ∈ dl}"

lemma preds_dl_union[simp]: "preds_dl (dl1 ∪ dl2) = preds_dl dl1 ∪ preds_dl dl2"
  ⟨proof⟩

type_synonym ('x,'c) var_val = "'x ⇒ 'c"

type_synonym ('p,'c) pred_val = "'p ⇒ 'c list set"


fun eval_id :: "('x,'c) id ⇒ ('x,'c) var_val ⇒ 'c" ("⟦_⟧id") where
  "⟦Var x⟧id σ = σ x"
| "⟦Cst c⟧id σ = c"

fun eval_ids :: "('x,'c) id list ⇒ ('x,'c) var_val ⇒ 'c list" ("⟦_⟧ids") where
  "⟦ids⟧ids σ = map (λa. ⟦a⟧id σ) ids"

fun meaning_rh :: "('p,'x,'c) rh ⇒ ('p,'c) pred_val ⇒ ('x,'c) var_val ⇒ bool" ("⟦_⟧rh") where
  "⟦a = a'⟧rh ϱ σ ⟷ ⟦a⟧id σ = ⟦a'⟧id σ"
| "⟦a ≠ a'⟧rh ϱ σ ⟷ ⟦a⟧id σ  ≠ ⟦a'⟧id σ"
| "⟦+ p ids⟧rh ϱ σ ⟷ ⟦ids⟧ids σ ∈ ϱ p"
| "⟦¬ p ids⟧rh ϱ σ ⟷ ¬ ⟦ids⟧ids σ ∈ ϱ p"

fun meaning_rhs :: "('p,'x,'c) rh list ⇒ ('p,'c) pred_val ⇒ ('x,'c) var_val ⇒ bool" ("⟦_⟧rhs") where
  "⟦rhs⟧rhs ϱ σ ⟷ (∀rh ∈ set rhs. ⟦rh⟧rh ϱ σ)"

fun meaning_lh :: "('p,'x,'c) lh ⇒ ('p,'c) pred_val ⇒ ('x,'c) var_val ⇒ bool" ("⟦_⟧lh") where
  "⟦(p,ids)⟧lh ϱ σ ⟷ ⟦ids⟧ids σ ∈ ϱ p"

fun meaning_cls :: "('p,'x,'c) clause ⇒ ('p,'c) pred_val ⇒ ('x,'c) var_val ⇒ bool" ("⟦_⟧cls") where
  "⟦Cls p ids rhs⟧cls ϱ σ ⟷ (⟦rhs⟧rhs ϱ σ ⟶ ⟦(p,ids)⟧lh ϱ σ)"

fun solves_lh :: "('p,'c) pred_val ⇒ ('p,'x,'c) lh ⇒ bool" (infix "⊨lh" 91) where
  "ϱ ⊨lh (p,ids) ⟷ (∀σ. ⟦(p,ids)⟧lh ϱ σ)"

fun solves_rh :: "('p,'c) pred_val ⇒ ('p,'x,'c) rh ⇒ bool" (infix "⊨rh" 91) where
  "ϱ ⊨rh rh ⟷ (∀σ. ⟦rh⟧rh ϱ σ)"
```

definition solves_cls :: "('p,'c) pred_val ⇒ ('p,'x,'c) clause ⇒ bool" (**infix** "$\models_{cls}$" 91) **where**
"$\varrho \models_{cls}$ c ⟷ (∀σ. ⟦c⟧$_{cls}$ ϱ σ)"

definition solves_program :: "('p,'c) pred_val ⇒ ('p,'x,'c) dl_program ⇒ bool" (**infix** "$\models_{dl}$" 91) **where**
"$\varrho \models_{dl}$ dl ⟷ (∀c ∈ dl. $\varrho \models_{cls}$ c)"

3 Substitutions

type_synonym ('x,'c) subst = "'x ⇒ ('x,'c) id"

fun subst_id :: "('x,'c) id ⇒ ('x,'c) subst ⇒ ('x,'c) id" (**infix** "$\cdot_{id}$" 70) **where**
"(Var x) $\cdot_{id}$ η = η x"
| "(Cst e) $\cdot_{id}$ η = (Cst e)"

fun subst_ids :: "('x,'c) id list ⇒ ('x,'c) subst ⇒ ('x,'c) id list" (**infix** "$\cdot_{ids}$" 50) **where**
"ids $\cdot_{ids}$ η = map (λa. a $\cdot_{id}$ η) ids"

fun subst_rh :: "('p,'x,'c) rh ⇒ ('x,'c) subst ⇒ ('p,'x,'c) rh" (**infix** "$\cdot_{rh}$" 50) **where**
"(a = a') $\cdot_{rh}$ η = (a $\cdot_{id}$ η = a' $\cdot_{id}$ η)"
| "(a ≠ a') $\cdot_{rh}$ η = (a $\cdot_{id}$ η ≠ a' $\cdot_{id}$ η)"
| "(+ p ids) $\cdot_{rh}$ η = (+ p (ids $\cdot_{ids}$ η))"
| "(¬ p ids) $\cdot_{rh}$ η = (¬ p (ids $\cdot_{ids}$ η))"

fun subst_rhs :: "('p,'x,'c) rh list ⇒ ('x,'c) subst ⇒ ('p,'x,'c) rh list" (**infix** "$\cdot_{rhs}$" 50) **where**
"rhs $\cdot_{rhs}$ η = map (λa. a $\cdot_{rh}$ η) rhs"

fun subst_lh :: "('p,'x,'c) lh ⇒ ('x,'c) subst ⇒ ('p,'x,'c) lh" (**infix** "$\cdot_{lh}$" 50) **where**
"(p,ids) $\cdot_{lh}$ η = (p, ids $\cdot_{ids}$ η)"

fun subst_cls :: "('p,'x,'c) clause ⇒ ('x,'c) subst ⇒ ('p,'x,'c) clause" (**infix** "$\cdot_{cls}$" 50) **where**
"(Cls p ids rhs) $\cdot_{cls}$ η = Cls p (ids $\cdot_{ids}$ η) (rhs $\cdot_{rhs}$ η)"

definition compose :: "('x,'c) subst ⇒ ('x,'c) var_val ⇒ ('x,'c) var_val" (**infix** "$\circ_{sv}$" 50) **where**
"(η $\circ_{sv}$ σ) x = ⟦(η x)⟧$_{id}$ σ"

4 Substituting variable valuations

fun substv_id :: "('x,'c) id ⇒ ('x,'c) var_val ⇒ ('x,'c) id" (**infix** "$\cdot_{vid}$" 70) **where**
"(Var x) $\cdot_{vid}$ σ = Cst (σ x)"
| "(Cst e) $\cdot_{vid}$ σ = (Cst e)"

fun substv_ids :: "('x,'c) id list ⇒ ('x,'c) var_val ⇒ ('x,'c) id list" (**infix** "$\cdot_{vids}$" 50) **where**
"ids $\cdot_{vids}$ σ = map (λa. a $\cdot_{vid}$ σ) ids"

fun substv_rh :: "('p,'x,'c) rh ⇒ ('x,'c) var_val ⇒ ('p,'x,'c) rh" (**infix** "$\cdot_{vrh}$" 50) **where**
"(a = a') $\cdot_{vrh}$ σ = (a $\cdot_{vid}$ σ = a' $\cdot_{vid}$ σ)"
| "(a ≠ a') $\cdot_{vrh}$ σ = (a $\cdot_{vid}$ σ ≠ a' $\cdot_{vid}$ σ)"
| "(+ p ids) $\cdot_{vrh}$ σ = (+ p (ids $\cdot_{vids}$ σ))"
| "(¬ p ids) $\cdot_{vrh}$ σ = (¬ p (ids $\cdot_{vids}$ σ))"

definition substv_rhs :: "('p,'x,'c) rh list ⇒ ('x,'c) var_val ⇒ ('p,'x,'c) rh list" (**infix** "$\cdot_{vrhs}$" 50) **where**
"rhs $\cdot_{vrhs}$ σ = map (λa. a $\cdot_{vrh}$ σ) rhs"

fun substv_lh :: "('p,'x,'c) lh ⇒ ('x,'c) var_val ⇒ ('p,'x,'c) lh" (**infix** "$\cdot_{vlh}$" 50) **where**
"(p,ids) $\cdot_{vlh}$ σ = (p, ids $\cdot_{vids}$ σ)"

fun substv_cls :: "('p,'x,'c) clause ⇒ ('x,'c) var_val ⇒ ('p,'x,'c) clause" (**infix** "$\cdot_{vcls}$" 50) **where**
"(Cls p ids rhs) $\cdot_{vcls}$ σ = Cls p (ids $\cdot_{vids}$ σ) (rhs $\cdot_{vrhs}$ σ)"

5 Datalog lemmas

5.1 Variable valuations

lemma substv_id_is_Cst_eval_id:
"a' $\cdot_{vid}$ σ' = Cst ($[\![$a'$]\!]_{id}$ σ')"
⟨*proof*⟩

lemma eval_id_is_substv_id:
"$[\![$a'$]\!]_{id}$ σ' = $[\![$a$]\!]_{id}$ σ $\longleftrightarrow$ (a' $\cdot_{vid}$ σ') = (a $\cdot_{vid}$ σ)"
⟨*proof*⟩

lemma eval_ids_is_substv_ids:
"$[\![$ids'$]\!]_{ids}$ σ' = $[\![$ids$]\!]_{ids}$ σ $\longleftrightarrow$ (ids' $\cdot_{vids}$ σ') = (ids $\cdot_{vids}$ σ)"
⟨*proof*⟩

lemma solves_rh_substv_rh_if_meaning_rh:
assumes "$[\![$a$]\!]_{rh}$ ϱ σ"
shows "ϱ $\models_{rh}$ (a $\cdot_{vrh}$ σ)"
⟨*proof*⟩

lemma solves_lh_substv_lh_if_meaning_lh:
assumes "$[\![$a$]\!]_{lh}$ ϱ σ"
shows "ϱ $\models_{lh}$ (a $\cdot_{vlh}$ σ)"
⟨*proof*⟩

5.2 Solve lhs

lemma solves_lh_iff_solves_lh: "ϱ $\models_{cls}$ Cls p ids [] $\longleftrightarrow$ ϱ $\models_{rh}$ (+ p ids)"
⟨*proof*⟩

lemma solves_lh_lh:
assumes "ϱ $\models_{cls}$ Cls p args []"
shows "ϱ $\models_{lh}$ (p, args)"
⟨*proof*⟩

lemmas solve_lhs = solves_lh_iff_solves_lh solves_lh_lh

5.3 Propositional inferences

5.3.1 Of last right hand

lemma prop_inf_last_from_cls_rh_to_cls:
assumes "ϱ $\models_{cls}$ Cls p ids (rhs @ [rh])"
assumes "ϱ $\models_{rh}$ rh"
shows "ϱ $\models_{cls}$ Cls p ids rhs"
⟨*proof*⟩

lemma prop_inf_last_from_cls_lh_to_cls:
assumes "ϱ $\models_{cls}$ Cls p ids (rhs @ [+ p ids])"
assumes "ϱ $\models_{lh}$ (p, ids)"
shows "ϱ $\models_{cls}$ Cls p ids rhs"
⟨*proof*⟩

lemmas prop_inf_last = prop_inf_last_from_cls_rh_to_cls prop_inf_last_from_cls_lh_to_cls

5.3.2 Of only right hand

lemma modus_ponens_rh:
assumes "ϱ $\models_{cls}$ Cls p ids [+ p' ids']"
assumes "ϱ $\models_{lh}$ (p', ids')"
shows "ϱ $\models_{lh}$ (p, ids)"
⟨*proof*⟩

lemma prop_inf_only_from_cls_cls_to_cls:

assumes "$\varrho \models_{cls}$ Cls p ids [+ p' ids']"
assumes "$\varrho \models_{cls}$ Cls p' ids' []"
shows "$\varrho \models_{cls}$ Cls p ids []"
⟨*proof*⟩

lemmas prop_inf_only = modus_ponens_rh prop_inf_only_from_cls_cls_to_cls

5.3.3 Of all right hands

lemma modus_ponens:
assumes "$\varrho \models_{cls}$ Cls p ids rhs"
assumes "$\forall$rh$\in$set rhs. $\varrho \models_{rh}$ rh"
shows "$\varrho \models_{lh}$ (p, ids)"
⟨*proof*⟩

lemmas prop_inf_all = modus_ponens

lemmas prop_inf = prop_inf_last prop_inf_only prop_inf_all

5.4 Substitution

lemma substitution_lemma_id: "$[\![$a$]\!]_{id}$ ($\eta \circ_{sv} \sigma$) = $[\![$a $\cdot_{id}$ $\eta]\!]_{id}$ σ"
⟨*proof*⟩

lemma substitution_lemma_ids: "$[\![$ids$]\!]_{ids}$ ($\eta \circ_{sv} \sigma$) = $[\![$ids $\cdot_{ids}$ $\eta]\!]_{ids}$ σ"
⟨*proof*⟩

lemma substitution_lemma_lh: "$[\![$(p, ids)$]\!]_{lh}$ ϱ ($\eta \circ_{sv} \sigma$) $\longleftrightarrow$ $[\![$(p, ids $\cdot_{ids}$ η)$]\!]_{lh}$ ϱ σ"
⟨*proof*⟩

lemma substitution_lemma_rh:"$[\![$rh$]\!]_{rh}$ ϱ ($\eta \circ_{sv} \sigma$) $\longleftrightarrow$ $[\![$rh $\cdot_{rh}$ $\eta]\!]_{rh}$ ϱ σ"
⟨*proof*⟩

lemma substitution_lemma_rhs: "$[\![$rhs$]\!]_{rhs}$ ϱ ($\eta \circ_{sv} \sigma$) $\longleftrightarrow$ $[\![$rhs $\cdot_{rhs}$ $\eta]\!]_{rhs}$ ϱ σ"
⟨*proof*⟩

lemma substitution_lemma_cls:
"$[\![$c$]\!]_{cls}$ ϱ ($\eta \circ_{sv} \sigma$) = $[\![$c $\cdot_{cls}$ $\eta]\!]_{cls}$ ϱ σ"
⟨*proof*⟩

lemma substitution_lemma:
assumes "$\varrho \models_{cls}$ c"
shows "$\varrho \models_{cls}$ (c $\cdot_{cls}$ (η::('x,'c) subst))"
⟨*proof*⟩

6 Stratification and solutions to stratified datalog programs

type_synonym 'p strat = "'p ⇒ nat"

fun rnk :: "'p strat ⇒ ('p,'x,'c) rh ⇒ nat" **where**
"rnk s (a = a') = 0"
| "rnk s (a ≠ a') = 0"
| "rnk s (+ p ids) = s p"
| "rnk s (¬ p ids) = 1 + s p"

fun strat_wf_cls :: "'p strat ⇒ ('p,'x,'c) clause ⇒ bool" **where**
"strat_wf_cls s (Cls p ids rhs) ⟷ (∀rh ∈ set rhs. s p ≥ rnk s rh)"

definition strat_wf :: "'p strat ⇒ ('p,'x,'c) dl_program ⇒ bool" **where**
"strat_wf s dl ⟷ (∀c ∈ dl. strat_wf_cls s c)"

definition max_stratum :: "'p strat ⇒ ('p,'x,'c) dl_program ⇒ nat" **where**
"max_stratum s dl = Max {s p | p ids rhs. Cls p ids rhs ∈ dl}"

fun pred_val_lte_stratum :: "('p,'c) pred_val ⇒ 'p strat ⇒ nat ⇒ ('p,'c) pred_val"
  ("_ ≤≤≤_≤≤≤ _" 0) **where**
  "(ϱ ≤≤≤s≤≤≤ n) p = (if s p ≤ n then ϱ p else {})"

fun dl_program_lte_stratum :: "('p,'x,'c) dl_program ⇒ 'p strat ⇒ nat ⇒ ('p,'x,'c) dl_program"
  ("_ ≤≤_≤≤ _" 0) **where**
  "(dl ≤≤s≤≤ n) = {(Cls p ids rhs)| p ids rhs . (Cls p ids rhs) ∈ dl ∧ s p ≤ n}"

fun dl_program_on_stratum :: "('p,'x,'c) dl_program ⇒ 'p strat ⇒ nat ⇒ ('p,'x,'c) dl_program"
  ("_ ==_== _" 0) **where**
  "(dl ==s== n) = {(Cls p ids rhs)| p ids rhs . (Cls p ids rhs) ∈ dl ∧ s p = n}"

— The ordering on predicate valuations from Flemming Nielson and Hanne Riis Nielson. Program analysis (an appetizer). CoRR,abs/2012.10086, 2020.
definition lt :: "('p,'c) pred_val ⇒ 'p strat ⇒ ('p,'c) pred_val ⇒ bool" ("_ ⊏_⊏ _") **where**
  "(ϱ ⊏s⊏ ϱ') ⟷ (∃p. ϱ p ⊂ ϱ' p ∧
                    (∀p'. s p' = s p ⟶ ϱ p' ⊆ ϱ' p') ∧
                    (∀p'. s p' < s p ⟶ ϱ p' = ϱ' p'))"

— The ordering on predicate valuations from Teodor C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, Foundations of Deductive Databases and Logic Programming, pages 193216. Morgan Kaufmann, 1988.
definition lt_prz :: "('p,'c) pred_val ⇒ 'p strat ⇒ ('p,'c) pred_val ⇒ bool" ("_ ⊏_⊏$_{prz}$ _") **where**
  "($ϱ_M$ ⊏s⊏$_{prz}$ $ϱ_N$) ⟷ $ϱ_N$ ≠ $ϱ_M$ ∧
                  (∀p_A c_A. c_A ∈ $ϱ_N$ p_A - $ϱ_M$ p_A ⟶ (∃p_B c_B. c_B ∈ $ϱ_M$ p_B - $ϱ_N$ p_B ∧ s p_A > s p_B))"

definition lte :: "('p,'c) pred_val ⇒ 'p strat ⇒ ('p,'c) pred_val ⇒ bool" ("_ ⊑_⊑ _") **where**
  "(ϱ ⊑s⊑ ϱ') ⟷ ϱ = ϱ' ∨ (ϱ ⊏s⊏ ϱ')"

definition least_solution :: "('p,'c) pred_val ⇒ ('p,'x,'c) dl_program ⇒ 'p strat ⇒ bool"
  ("_ ⊨$_{lst}$") **where**
  "ϱ ⊨$_{lst}$ dl s ⟷ ϱ ⊨$_{dl}$ dl ∧ (∀ϱ'. ϱ' ⊨$_{dl}$ dl ⟶ ϱ ⊑s⊑ ϱ')"

definition minimal_solution :: "('p,'c) pred_val ⇒ ('p,'x,'c) dl_program ⇒ 'p strat ⇒ bool"
  ("_ ⊨$_{min}$") **where**
  "ϱ ⊨$_{min}$ dl s ⟷ ϱ ⊨$_{dl}$ dl ∧ (∄ϱ'. ϱ' ⊨$_{dl}$ dl ∧ ϱ' ⊏s⊏ ϱ)"

lemma lte_def2:
  "(ϱ ⊑s⊑ ϱ') ⟷ ϱ ≠ ϱ' ⟶ (ϱ ⊏s⊏ ϱ')"
  ⟨*proof*⟩

6.1 Solving lower strata

lemma strat_wf_lte_if_strat_wf_lte:
  assumes "n > m"
  assumes "strat_wf s (dl ≤≤s≤≤ n)"
  shows "strat_wf s (dl ≤≤s≤≤ m)"
  ⟨*proof*⟩

lemma strat_wf_lte_if_strat_wf:
  assumes "strat_wf s dl"
  shows "strat_wf s (dl ≤≤s≤≤ n)"
  ⟨*proof*⟩

lemma meaning_lte_m_iff_meaning_rh:
  assumes "rnk s rh ≤ n"
  shows "⟦rh⟧$_{rh}$ (ϱ ≤≤≤s≤≤≤ n) σ ⟷ ⟦rh⟧$_{rh}$ ϱ σ"
  ⟨*proof*⟩

lemma meaning_lte_m_iff_meaning_lh:
  assumes "s p ≤ m"
  shows "⟦(p, ids)⟧$_{lh}$ (ϱ ≤≤≤s≤≤≤ m) σ ⟷ ⟦(p, ids)⟧$_{lh}$ ϱ σ"
  ⟨*proof*⟩

lemma meaning_lte_m_iff_meaning_cls:
  assumes "strat_wf_cls s (Cls p ids rhs)"
  assumes "s p ≤ m"
  shows "⟦Cls p ids rhs⟧$_{cls}$ (ϱ ≤≤≤s≤≤≤ m) σ ⟷ ⟦Cls p ids rhs⟧$_{cls}$ ϱ σ"
⟨*proof*⟩

lemma solves_lte_m_iff_solves_cls:
  assumes "strat_wf_cls s (Cls p ids rhs)"
  assumes "s p ≤ m"
  shows "(ϱ ≤≤≤s≤≤≤ m) ⊨$_{cls}$ Cls p ids rhs ⟷ ϱ ⊨$_{cls}$ Cls p ids rhs"
  ⟨*proof*⟩

lemma downward_lte_solves:
  assumes "n > m"
  assumes "ϱ ⊨$_{dl}$ (dl ≤≤s≤≤ n)"
  assumes "strat_wf s dl"
  shows "(ϱ ≤≤≤s≤≤≤ m) ⊨$_{dl}$ (dl ≤≤s≤≤ m)"
  ⟨*proof*⟩

lemma downward_solves:
  assumes "ϱ ⊨$_{dl}$ dl"
  assumes "strat_wf s dl"
  shows "(ϱ ≤≤≤s≤≤≤ m) ⊨$_{dl}$ (dl ≤≤s≤≤ m)"
  ⟨*proof*⟩

6.2 Least solutions

6.2.1 Existence of least solutions

definition Inter' :: "('a ⇒ 'b set) set ⇒ 'a ⇒ 'b set" ("⋂") **where**
  "(⋂ ϱs) = (λp. ⋂{ϱ p | ϱ. ϱ ∈ ϱs})"

lemma Inter'_def2: "(⋂ ϱs) = (λp. ⋂{m. ∃ϱ ∈ ϱs. m = ϱ p})"
  ⟨*proof*⟩

lemma member_Inter':
  assumes "∀p ∈ ps. y ∈ p x"
  shows "y ∈ (⋂ ps) x"
  ⟨*proof*⟩

lemma member_if_member_Inter':
  assumes "y ∈ (⋂ ps) x"
  assumes "p ∈ ps"
  shows "y ∈ p x"
  ⟨*proof*⟩

lemma member_Inter'_iff:
  "(∀p ∈ ps. y ∈ p x) ⟷ y ∈ (⋂ ps) x"
  ⟨*proof*⟩

lemma intersection_valuation_subset_valuation:
  assumes "P ϱ"
  shows "⋂ {ϱ'. P ϱ'} p ⊆ ϱ p"
  ⟨*proof*⟩

fun solve_pg **where**
  "solve_pg s dl 0 = (⋂ {ϱ. ϱ ⊨$_{dl}$ (dl ==s== 0)})"
| "solve_pg s dl (Suc n) = (⋂ {ϱ. ϱ ⊨$_{dl}$ (dl ==s== Suc n) ∧ (ϱ ≤≤≤s≤≤≤ n) = solve_pg s dl n})"

definition least_rank_p_st :: "('p ⇒ bool) ⇒ 'p ⇒ ('p ⇒ nat) ⇒ bool" **where**
  "least_rank_p_st P p s ⟷ P p ∧ (∀p'. P p' ⟶ s p ≤ s p')"

lemma least_rank_p_st_exists:

assumes "P p"
shows "∃p''. least_rank_p_st P p'' s"
⟨*proof*⟩

lemma below_least_rank_p_st:
assumes "least_rank_p_st P p'' s"
assumes "s p < s p''"
shows "¬P p"
⟨*proof*⟩

lemma lt_if_lt_prz:
assumes "ϱ_M ⊏s⊏$_{prz}$ ϱ_N"
shows "ϱ_N ⊏s⊏ ϱ_M"
⟨*proof*⟩

lemma lt_prz_if_lt_if:
assumes "ϱ_M ⊏s⊏ ϱ_N"
shows "ϱ_N ⊏s⊏$_{prz}$ ϱ_M"
⟨*proof*⟩

lemma lt_prz_iff_lt:
"ϱ_M ⊏s⊏ ϱ_N ⟷ ϱ_N ⊏s⊏$_{prz}$ ϱ_M"
⟨*proof*⟩

lemma solves_leq:
assumes "ϱ ⊨$_{dl}$ (dl ≤≤s≤≤ m)"
assumes "n ≤ m"
shows "ϱ ⊨$_{dl}$ (dl ≤≤s≤≤ n)"
⟨*proof*⟩

lemma solves_Suc:
assumes "ϱ ⊨$_{dl}$ (dl ≤≤s≤≤ Suc n)"
shows "ϱ ⊨$_{dl}$ (dl ≤≤s≤≤ n)"
⟨*proof*⟩

lemma solve_pg_0_subset:
assumes "ϱ ⊨$_{dl}$ (dl ≤≤s≤≤ 0)"
shows "(solve_pg s dl 0) p ⊆ ϱ p"
⟨*proof*⟩

lemma solve_pg_Suc_subset:
assumes "ϱ ⊨$_{dl}$ (dl ==s== Suc n)"
assumes "(ϱ ≤≤≤s≤≤≤ n) = solve_pg s dl n"
shows "(solve_pg s dl (Suc n)) p ⊆ ϱ p"
⟨*proof*⟩

lemma solve_pg_0_empty:
assumes "s p > 0"
shows "(solve_pg s dl 0) p = {}"
⟨*proof*⟩

lemma solve_pg_above_empty:
assumes "s p > n"
shows "(solve_pg s dl n) p = {}"
⟨*proof*⟩

lemma exi_sol_n:
"∃ϱ'. ϱ' ⊨$_{dl}$ (dl ==s== Suc m) ∧ (ϱ' ≤≤≤s≤≤≤ m) = solve_pg s dl m"
⟨*proof*⟩

lemma solve_pg_agree_above:
assumes "s p ≤ m"
shows "(solve_pg s dl m) p = (solve_pg s dl (s p)) p"

⟨*proof*⟩

lemma solve_pg_two_agree_above:
assumes "s p ≤ n"
assumes "s p ≤ m"
shows "(solve_pg s dl m) p = (solve_pg s dl n) p"
⟨*proof*⟩

lemma pos_rhs_stratum_leq_clause_stratum:
assumes "strat_wf s dl"
assumes "Cls p ids rhs ∈ dl"
assumes "+ p' ids' ∈ set rhs"
shows "s p' ≤ s p"
⟨*proof*⟩

lemma neg_rhs_stratum_less_clause_stratum:
assumes "strat_wf s dl"
assumes "Cls p ids rhs ∈ dl"
assumes "¬ p' ids' ∈ set rhs"
shows "s p' < s p"
⟨*proof*⟩

lemma solve_pg_two_agree_above_on_rh:
assumes "strat_wf s dl"
assumes "Cls p ids rhs ∈ dl"
assumes "s p ≤ n"
assumes "s p ≤ m"
assumes "rh ∈ set rhs"
shows "$[\![$rh$]\!]_{rh}$ (solve_pg s dl m) σ ⟷ $[\![$rh$]\!]_{rh}$ (solve_pg s dl n) σ"
⟨*proof*⟩

lemma solve_pg_two_agree_above_on_lh:
assumes "s p ≤ n"
assumes "s p ≤ m"
shows "$[\![$(p,ids)$]\!]_{lh}$ (solve_pg s dl m) σ ⟷ $[\![$(p,ids)$]\!]_{lh}$ (solve_pg s dl n) σ"
⟨*proof*⟩

lemma solve_pg_two_agree_above_on_cls:
assumes "strat_wf s dl"
assumes "Cls p ids rhs ∈ dl"
assumes "s p ≤ n"
assumes "s p ≤ m"
shows "$[\![$Cls p ids rhs$]\!]_{cls}$ (solve_pg s dl n) σ ⟷ $[\![$Cls p ids rhs$]\!]_{cls}$ (solve_pg s dl m) σ"
⟨*proof*⟩

lemma solve_pg_two_agree_above_on_cls_Suc:
assumes "strat_wf s dl"
assumes "Cls p ids rhs ∈ dl"
assumes "s p ≤ n"
shows "$[\![$Cls p ids rhs$]\!]_{cls}$ (solve_pg s dl (Suc n)) σ ⟷ $[\![$Cls p ids rhs$]\!]_{cls}$ (solve_pg s dl n) σ"
⟨*proof*⟩

lemma stratum0_no_neg':
assumes "strat_wf s dl"
assumes "Cls p ids rhs ∈ dl"
assumes "s p = 0"
assumes "rh ∈ set rhs"
shows "∄p' ids. rh = ¬ p' ids"
⟨*proof*⟩

lemma stratumSuc_less_neg':
assumes "strat_wf s dl"
assumes "Cls p ids rhs ∈ dl"

assumes "s p = Suc n"
assumes "¬ p' ids' ∈ set rhs"
shows "s p' ≤ n"
⟨*proof*⟩

lemma stratum0_no_neg:
assumes "strat_wf s dl"
assumes "Cls p ids rhs ∈ (dl ≤≤s≤≤ 0)"
assumes "rh ∈ set rhs"
shows "∄p' ids. rh = ¬ p ids"
⟨*proof*⟩

lemma stratumSuc_less_neg:
assumes "strat_wf s dl"
assumes "Cls p ids rhs ∈ (dl ≤≤s≤≤ Suc n)"
assumes "¬ p' ids' ∈ set rhs"
shows "s p' ≤ n"
⟨*proof*⟩

lemma all_meaning_rh_if_solve_pg_0:
assumes "strat_wf s dl"
assumes "⟦rh⟧$_{rh}$ (solve_pg s dl 0) σ"
assumes "ϱ ⊨$_{dl}$ (dl ≤≤s≤≤ 0)"
assumes "rh ∈ set rhs"
assumes "Cls p ids rhs ∈ (dl ≤≤s≤≤ 0)"
shows "⟦rh⟧$_{rh}$ ϱ σ"
⟨*proof*⟩

lemma all_meaning_rh_if_solve_pg_Suc:
assumes "strat_wf s dl"
assumes "⟦rh⟧$_{rh}$ (solve_pg s dl (Suc n)) σ"
assumes "ϱ ⊨$_{dl}$ (dl ==s== Suc n)"
assumes "(ϱ ≤≤≤s≤≤≤ n) = solve_pg s dl n"
assumes "rh ∈ set rhs"
assumes "Cls p ids rhs ∈ (dl ≤≤s≤≤ Suc n)"
shows "⟦rh⟧$_{rh}$ ϱ σ"
⟨*proof*⟩

lemma solve_pg_0_if_all_meaning_lh:
assumes "∀ϱ'. ϱ' ⊨$_{dl}$ (dl ≤≤s≤≤ 0) ⟶ ⟦(p, ids)⟧$_{lh}$ ϱ' σ"
shows "⟦(p, ids)⟧$_{lh}$ (solve_pg s dl 0) σ"
⟨*proof*⟩

lemma all_meaning_lh_if_solve_pg_0:
assumes "⟦(p, ids)⟧$_{lh}$ (solve_pg s dl 0) σ"
shows "∀ϱ'. ϱ' ⊨$_{dl}$ (dl ≤≤s≤≤ 0) ⟶ ⟦(p, ids)⟧$_{lh}$ ϱ' σ"
⟨*proof*⟩

lemma solve_pg_0_iff_all_meaning_lh:
"⟦(p, ids)⟧$_{lh}$ (solve_pg s dl 0) σ ⟷ (∀ϱ'. ϱ' ⊨$_{dl}$ (dl ≤≤s≤≤ 0) ⟶ ⟦(p, ids)⟧$_{lh}$ ϱ' σ)"
⟨*proof*⟩

lemma solve_pg_Suc_if_all_meaning_lh:
assumes "∀ϱ'. ϱ' ⊨$_{dl}$ (dl ==s== Suc n) ⟶ (ϱ' ≤≤≤s≤≤≤ n) = solve_pg s dl n ⟶ ⟦(p, ids)⟧$_{lh}$ ϱ' σ"
shows "⟦(p, ids)⟧$_{lh}$ (solve_pg s dl (Suc n)) σ"
⟨*proof*⟩

lemma all_meaning_if_solve_pg_Suc_lh:
assumes "⟦(p, ids)⟧$_{lh}$ (solve_pg s dl (Suc n)) σ"
shows "∀ϱ'. ϱ' ⊨$_{dl}$ (dl ==s== Suc n) ⟶ (ϱ' ≤≤≤s≤≤≤ n) = solve_pg s dl n ⟶ ⟦(p, ids)⟧$_{lh}$ ϱ' σ"
⟨*proof*⟩

lemma solve_pg_Suc_iff_all_meaning_lh:
"($[\![$(p, ids)$]\!]_{lh}$ (solve_pg s dl (Suc n)) σ) $\longleftrightarrow$
($\forall \varrho$'. ϱ' $\models_{dl}$ (dl ==s== Suc n) $\longrightarrow$ (ϱ' $\leq\leq\leq$s$\leq\leq\leq$ n) = solve_pg s dl n $\longrightarrow$ $[\![$(p, ids)$]\!]_{lh}$ ϱ' σ)"
⟨*proof*⟩

lemma solve_pg_0_meaning_cls':
assumes "strat_wf s dl"
assumes "Cls p ids rhs $\in$ (dl $\leq\leq$s$\leq\leq$ 0)"
shows "$[\![$Cls p ids rhs$]\!]_{cls}$ (solve_pg s dl 0) σ"
⟨*proof*⟩

lemma solve_pg_meaning_cls':
assumes "strat_wf s dl"
assumes "Cls p ids rhs $\in$ (dl $\leq\leq$s$\leq\leq$ n)"
shows "$[\![$Cls p ids rhs$]\!]_{cls}$ (solve_pg s dl n) σ"
⟨*proof*⟩

lemma solve_pg_meaning_cls:
assumes "strat_wf s dl"
assumes "c $\in$ (dl $\leq\leq$s$\leq\leq$ n)"
shows "$[\![$c$]\!]_{cls}$ (solve_pg s dl n) σ"
⟨*proof*⟩

lemma solve_pg_solves_cls:
assumes "strat_wf s dl"
assumes "c $\in$ (dl $\leq\leq$s$\leq\leq$ n)"
shows "solve_pg s dl n $\models_{cls}$ c"
⟨*proof*⟩

lemma solve_pg_solves_dl:
assumes "strat_wf s dl"
shows "solve_pg s dl n $\models_{dl}$ (dl $\leq\leq$s$\leq\leq$ n)"
⟨*proof*⟩

lemma disjE3:
assumes major: "P $\vee$ Q $\vee$ Z"
and minorP: "P $\Longrightarrow$ R"
and minorQ: "Q $\Longrightarrow$ R"
and minorZ: "Z $\Longrightarrow$ R"
shows R
⟨*proof*⟩

lemma solve_pg_0_below_solution:
assumes "ϱ $\models_{dl}$ (dl $\leq\leq$s$\leq\leq$ 0)"
shows "(solve_pg s dl 0) $\sqsubseteq$s$\sqsubseteq$ ϱ"
⟨*proof*⟩

lemma least_disagreement_proper_subset:
assumes "ϱ''n $\sqsubset$s$\sqsubset$ ϱ"
assumes "least_rank_p_st (λp. ϱ''n p $\neq$ ϱ p) p s"
shows "ϱ''n p $\subset$ ϱ p"
⟨*proof*⟩

lemma subset_on_least_disagreement:
assumes "ϱ''n $\sqsubset$s$\sqsubset$ ϱ"
assumes "least_rank_p_st (λp. ϱ''n p $\neq$ ϱ p) p s"
assumes "s p' = s p"
shows "ϱ''n p' $\subseteq$ ϱ p'"
⟨*proof*⟩

lemma equal_below_least_disagreement:
assumes "ϱ''n $\sqsubset$s$\sqsubset$ ϱ"
assumes "least_rank_p_st (λp. ϱ''n p $\neq$ ϱ p) p s"

assumes "s p' < s p"
shows "ϱ''n p' = ϱ p'"
⟨*proof*⟩

lemma solution_on_subset_solution_below:
"(dl ==s== n) ⊆ (dl ≤≤s≤≤ n)"
⟨*proof*⟩

lemma solves_program_mono:
assumes "dl ⊆ dl'"
assumes "ϱ $\models_{dl}$ dl'"
shows "ϱ $\models_{dl}$ dl"
⟨*proof*⟩

lemma solution_on_if_solution_below:
assumes "ϱ $\models_{dl}$ (dl ≤≤s≤≤ n)"
shows "ϱ $\models_{dl}$ (dl ==s== n)"
⟨*proof*⟩

lemma solve_pg_Suc_subset_solution:
assumes "ϱ $\models_{dl}$ (dl ≤≤s≤≤ Suc n)"
assumes "(ϱ ≤≤≤s≤≤≤ n) = solve_pg s dl n"
shows "solve_pg s dl (Suc n) p ⊆ ϱ p"
⟨*proof*⟩

lemma solve_pg_subset_solution:
assumes "m > n"
assumes "ϱ $\models_{dl}$ (dl ≤≤s≤≤ m)"
assumes "(ϱ ≤≤≤s≤≤≤ n) = solve_pg s dl n"
shows "solve_pg s dl (Suc n) p ⊆ ϱ p"
⟨*proof*⟩

lemma below_least_disagreement:
assumes "least_rank_p_st (λp. ϱ' p ≠ ϱ p) p s"
assumes "s p' < s p"
shows "ϱ' p' = ϱ p'"
⟨*proof*⟩

definition agree_below_eq :: "('p,'c) pred_val ⇒ ('p,'c) pred_val ⇒ nat ⇒ 'p strat ⇒ bool" **where**
"agree_below_eq ϱ ϱ' n s ⟷ (∀p. s p ≤ n ⟶ ϱ p = ϱ' p)"

definition agree_below :: "('p,'c) pred_val ⇒ ('p,'c) pred_val ⇒ nat ⇒ 'p strat ⇒ bool" **where**
"agree_below ϱ ϱ' n s ⟷ (∀p. s p < n ⟶ ϱ p = ϱ' p)"

definition agree_above :: "('p,'c) pred_val ⇒ ('p,'c) pred_val ⇒ nat ⇒ 'p strat ⇒ bool" **where**
"agree_above ϱ ϱ' n s ⟷ (∀p. s p > n ⟶ ϱ p = ϱ' p)"

definition agree_above_eq :: "('p,'c) pred_val ⇒ ('p,'c) pred_val ⇒ nat ⇒ 'p strat ⇒ bool" **where**
"agree_above_eq ϱ ϱ' n s ⟷ (∀p. s p ≥ n ⟶ ϱ p = ϱ' p)"

lemma agree_below_trans:
assumes "agree_below_eq ϱ ϱ' n s"
assumes "agree_below_eq ϱ' ϱ'' n s"
shows "agree_below_eq ϱ ϱ'' n s"
⟨*proof*⟩

lemma agree_below_eq_less_eq:
assumes "l ≤ n"
assumes "agree_below_eq ϱ ϱ' n s"
shows "agree_below_eq ϱ ϱ' l s"
⟨*proof*⟩

lemma agree_below_trans':

assumes "agree_below_eq ϱ ϱ' n s"
assumes "agree_below_eq ϱ' ϱ'' m s"
assumes "l $\leq$ n"
assumes "l $\leq$ m"
shows "agree_below_eq ϱ ϱ'' l s"
⟨*proof*⟩

lemma agree_below_eq_least_disagreement:
assumes "least_rank_p_st (λp. ϱ' p $\neq$ ϱ p) p s"
assumes "n < s p"
shows "agree_below_eq ϱ' ϱ n s"
⟨*proof*⟩

lemma agree_below_least_disagreement:
assumes "least_rank_p_st (λp. ϱ' p $\neq$ ϱ p) p s"
shows "agree_below ϱ' ϱ (s p) s"
⟨*proof*⟩

lemma eq_if_agree_below_eq_agree_above:
assumes "agree_below_eq ϱ ϱ' n s"
assumes "agree_above ϱ ϱ' n s"
shows "ϱ = ϱ'"
⟨*proof*⟩

lemma eq_if_agree_below_agree_above_eq:
assumes "agree_below ϱ ϱ' n s"
assumes "agree_above_eq ϱ ϱ' n s"
shows "ϱ = ϱ'"
⟨*proof*⟩

lemma eq_if_agree_below_eq_agree_above_eq:
assumes "agree_below_eq ϱ ϱ' n s"
assumes "agree_above_eq ϱ ϱ' n s"
shows "ϱ = ϱ'"
⟨*proof*⟩

lemma agree_below_eq_pred_val_lte_stratum:
"agree_below_eq ϱ (ϱ $\leq\leq\leq$s$\leq\leq\leq$ n) n s"
⟨*proof*⟩

lemma agree_below_eq_pred_val_lte_stratum_less_eq:
assumes "m $\leq$ n"
shows "agree_below_eq ϱ (ϱ $\leq\leq\leq$s$\leq\leq\leq$ n) m s"
⟨*proof*⟩

lemma agree_below_eq_solve_pg:
assumes "l $\leq$ m"
assumes "l $\leq$ n"
shows "agree_below_eq (solve_pg s dl n) (solve_pg s dl m) l s"
⟨*proof*⟩

lemma solve_pg_below_solution:
assumes "ϱ $\models_{dl}$ (dl $\leq\leq$s$\leq\leq$ n)"
shows "solve_pg s dl n $\sqsubseteq$s$\sqsubseteq$ ϱ"
⟨*proof*⟩

lemma solve_pg_least_solution':
assumes "strat_wf s dl"
shows "solve_pg s dl n $\models_{lst}$ (dl $\leq\leq$s$\leq\leq$ n) s"
⟨*proof*⟩

lemma stratum_less_eq_max_stratum:

assumes "finite dl"
assumes "Cls p ids rhs ∈ dl"
shows "s p ≤ max_stratum s dl"
⟨*proof*⟩

lemma finite_above_max_stratum:
assumes "finite dl"
assumes "max_stratum s dl ≤ n"
shows "(dl ≤≤s≤≤ n) = dl"
⟨*proof*⟩

lemma finite_max_stratum:
assumes "finite dl"
shows "(dl ≤≤s≤≤ max_stratum s dl) = dl"
⟨*proof*⟩

lemma solve_pg_least_solution:
assumes "finite dl"
assumes "strat_wf s dl"
shows "solve_pg s dl (max_stratum s dl) ⊨$_{lst}$ dl s"
⟨*proof*⟩

lemma exi_least_solution:
assumes "finite dl"
assumes "strat_wf s dl"
shows "∃ϱ. ϱ ⊨$_{lst}$ dl s"
⟨*proof*⟩

6.2.2 Equality of least and minimal solution

lemma least_iff_minimal:
assumes "finite dl"
assumes "strat_wf s dl"
shows "ϱ ⊨$_{lst}$ dl s ⟷ ϱ ⊨$_{min}$ dl s"
⟨*proof*⟩

6.2.3 Least solution on lower strata

lemma below_subset:
"(dl ≤≤s≤≤ n) ⊆ dl"
⟨*proof*⟩

lemma finite_below_finite:
assumes "finite dl"
shows "finite (dl ≤≤s≤≤ n)"
⟨*proof*⟩

lemma downward_least_solution:
assumes "finite dl"
assumes "n > m"
assumes "strat_wf s dl"
assumes "ϱ ⊨$_{lst}$ (dl ≤≤s≤≤ n) s"
shows "(ϱ ≤≤≤s≤≤≤ m) ⊨$_{lst}$ (dl ≤≤s≤≤ m) s"
⟨*proof*⟩

lemma downward_least_solution_same_stratum:
assumes "finite dl"
assumes "strat_wf s dl"
assumes "ϱ ⊨$_{lst}$ dl s"
shows "(ϱ ≤≤≤s≤≤≤ m) ⊨$_{lst}$ (dl ≤≤s≤≤ m) s"
⟨*proof*⟩

6.3 Negation

definition agree_var_val :: "'x set ⇒ ('x, 'c) var_val ⇒ ('x, 'c) var_val ⇒ bool " **where**
  "agree_var_val xs σ σ' ⟷ (∀x ∈ xs. σ x = σ' x)"

fun vars_ids :: "('a, 'b) id list ⇒ 'a set" **where**
  "vars_ids ids = ⋃(vars_id ‘ set ids)"

fun vars_lh :: "('p,'x,'c) lh ⇒ 'x set" **where**
  "vars_lh (p,ids) = vars_ids ids"

definition lh_consequence :: "('p, 'c) pred_val ⇒ ('p, 'x, 'c) clause ⇒ ('p, 'x, 'c) lh ⇒ bool"
  where
  "lh_consequence ϱ c lh ⟷ (∃σ'. ((the_lh c) $\cdot_{vlh}$ σ') = lh ∧ ⟦the_rhs c⟧$_{rhs}$ ϱ σ')"

lemma meaning_rh_iff_meaning_rh_pred_val_lte_stratum:
  assumes "c ∈ (dl ≤≤s≤≤ s p)"
  assumes "strat_wf s dl"
  assumes "rh ∈ set (the_rhs c)"
  shows "⟦rh⟧$_{rh}$ ϱ σ' ⟷ ⟦rh⟧$_{rh}$ (ϱ ≤≤≤s≤≤≤ s p) σ'"
⟨*proof*⟩

lemma meaning_rhs_iff_meaning_rhs_pred_val_lte_stratum:
  assumes "c ∈ (dl ≤≤s≤≤ s p)"
  assumes "strat_wf s dl"
  shows "⟦the_rhs c⟧$_{rhs}$ ϱ σ' ⟷ ⟦the_rhs c⟧$_{rhs}$ (ϱ ≤≤≤s≤≤≤ s p) σ'"
  ⟨*proof*⟩

lemma meaning_rhs_if_meaning_rhs_with_removed_top_strata:
  assumes "⟦rhs⟧$_{rhs}$ (ϱ'(p := ϱ' p - {⟦ids⟧$_{ids}$ σ})) σ'"
  assumes "strat_wf s dl"
  assumes c_dl': "Cls p' ids' rhs ∈ (dl ≤≤s≤≤ s p)"
  shows "⟦rhs⟧$_{rhs}$ ϱ' σ'"
⟨*proof*⟩

lemma meaning_PosLit_least':
  assumes "finite dl"
  assumes "ϱ ⊨$_{lst}$ dl s"
  assumes "strat_wf s dl"
  assumes "⟦+ p ids⟧$_{rh}$ ϱ σ"
  shows "∃c ∈ dl. lh_consequence ϱ c ((p,ids) $\cdot_{vlh}$ σ)"
⟨*proof*⟩

lemma meaning_lh_least':
  assumes "finite dl"
  assumes "ϱ ⊨$_{lst}$ dl s"
  assumes "strat_wf s dl"
  assumes "⟦(p, ids)⟧$_{lh}$ ϱ σ"
  shows "∃c ∈ dl. lh_consequence ϱ c ((p,ids) $\cdot_{vlh}$ σ)"
  ⟨*proof*⟩

lemma meaning_lh_least:
  assumes "finite dl"
  assumes "ϱ ⊨$_{lst}$ dl s"
  assumes "strat_wf s dl"
  shows "⟦(p, ids)⟧$_{lh}$ ϱ σ ⟷ (∃c ∈ dl. lh_consequence ϱ c ((p,ids) $\cdot_{vlh}$ σ))"
⟨*proof*⟩

lemma meaning_PosLit_least:
  assumes "finite dl"
  assumes "ϱ ⊨$_{lst}$ dl s"
  assumes "strat_wf s dl"
  shows "⟦+ p ids⟧$_{rh}$ ϱ σ ⟷ (∃c ∈ dl. lh_consequence ϱ c ((p,ids) $\cdot_{vlh}$ σ))"
⟨*proof*⟩

lemma meaning_NegLit_least:
  assumes "finite dl"
  assumes "ϱ $\models_{lst}$ dl s"
  assumes "strat_wf s dl"
  shows "$[\![\neg$ p ids$]\!]_{rh}$ ϱ σ $\longleftrightarrow$ ($\neg$($\exists$c $\in$ dl. lh_consequence ϱ c ((p,ids) $\cdot_{vlh}$ σ)))"
  ⟨*proof*⟩

lemma solves_PosLit_least:
  assumes "finite dl"
  assumes "ϱ $\models_{lst}$ dl s"
  assumes "strat_wf s dl"
  assumes "$\forall$a $\in$ set ids. is_Cst a"
  shows "ϱ $\models_{rh}$ (+ p ids) $\longleftrightarrow$ ($\exists$c $\in$ dl. lh_consequence ϱ c (p,ids))"
⟨*proof*⟩

lemma solves_lh_least:
  assumes "finite dl"
  assumes "ϱ $\models_{lst}$ dl s"
  assumes "strat_wf s dl"
  assumes "$\forall$a $\in$ set ids. is_Cst a"
  shows "ϱ $\models_{lh}$ (p, ids) $\longleftrightarrow$ ($\exists$c $\in$ dl. lh_consequence ϱ c (p,ids))"
⟨*proof*⟩

lemma solves_NegLit_least:
  assumes "finite dl"
  assumes "ϱ $\models_{lst}$ dl s"
  assumes "strat_wf s dl"
  assumes "$\forall$a $\in$ set ids. is_Cst a"
  shows "ϱ $\models_{rh}$ ($\neg$ p ids) $\longleftrightarrow$ $\neg$($\exists$c $\in$ dl. lh_consequence ϱ c (p,ids))"
⟨*proof*⟩

end
theory Program_Graph **imports** Labeled_Transition_Systems.LTS Datalog **begin**

7 Actions

datatype (fv_arith: 'v) arith =
  Integer int
  | Arith_Var 'v
  | Arith_Op "'v arith" "int ⇒ int ⇒ int" "'v arith"
  | Minus "'v arith"

datatype (fv_boolean: 'v) boolean =
  true
  | false
  | Rel_Op "'v arith" "int ⇒ int ⇒ bool" "'v arith"
  | Bool_Op "'v boolean" "bool ⇒ bool ⇒ bool" "'v boolean"
  | Neg "'v boolean"

datatype 'v action =
  Asg 'v "'v arith" ("_ ::= _" [1000, 61] 61)
  | Bool "'v boolean"
  | Skip

8 Memories

type_synonym 'v memory = "'v ⇒ int"

9 Semantics

fun sem_arith :: "'v arith ⇒ 'v memory ⇒ int" **where**

```
  "sem_arith (Integer n) σ = n"
| "sem_arith (Arith_Var x) σ = σ x"
| "sem_arith (Arith_Op a1 op a2) σ = op (sem_arith a1 σ) (sem_arith a2 σ)"
| "sem_arith (Minus a) σ = - (sem_arith a σ)"

fun sem_bool :: "'v boolean ⇒ 'v memory ⇒ bool" where
  "sem_bool true σ = True"
| "sem_bool false σ = False"
| "sem_bool (Rel_Op a1 op a2) σ = op (sem_arith a1 σ) (sem_arith a2 σ)"
| "sem_bool (Bool_Op b1 op b2) σ = op (sem_bool b1 σ) (sem_bool b2 σ)"
| "sem_bool (Neg b) σ = (¬(sem_bool b σ))"

fun sem_action :: "'v action ⇒ 'v memory ⇀ 'v memory" where
  "sem_action (x ::= a) σ = Some (σ(x := sem_arith a σ))"
| "sem_action (Bool b) σ = (if sem_bool b σ then Some σ else None)"
| "sem_action Skip σ = Some σ"
```

10 Program Graphs

10.1 Types

```
type_synonym ('n,'a) edge = "'n × 'a × 'n"

type_synonym ('n,'a) program_graph = "('n,'a) edge set × 'n × 'n"

type_synonym ('n,'v) config = "'n * 'v memory"
```

10.2 Program Graph Locale

```
locale program_graph =
  fixes pg :: "('n,'a) program_graph"
begin

definition edges where
  "edges = fst pg"

definition start where
  "start = fst (snd pg)"

definition "end" where
  "end = snd (snd pg)"

definition pg_rev :: "('n,'a) program_graph" where
  "pg_rev = (rev_edge ‘ edges, end, start)"

end
```

10.3 Finite Program Graph Locale

```
locale finite_program_graph = program_graph pg
  for pg :: "('n::finite,'v) program_graph" +
  assumes "finite edges"
begin

lemma finite_pg_rev: "finite (fst pg_rev)"
  ⟨proof⟩

end

locale finite_action_program_graph =
  fixes pg :: "('n,'v action) program_graph"
begin
```

interpretation program_graph pg ⟨*proof*⟩

fun initial_config_of :: "('n,'v) config ⇒ bool" **where**
  "initial_config_of (q,σ) ⟷ q = start"

fun final_config_of :: "('n,'v) config ⇒ bool" **where**
  "final_config_of (q,σ) ⟷ q = end"

inductive exe_step :: "('n,'v) config ⇒ 'v action ⇒ ('n,'v) config ⇒ bool" **where**
  "(q1, α, q2) ∈ edges ⟹ sem_action α σ = Some σ' ⟹ exe_step (q1,σ) α (q2,σ')"

end

end
theory Bit_Vector_Framework **imports** Program_Graph **begin**

— We encode the Bit Vector Framework into Datalog.

11 Bit-Vector Framework

11.1 Definitions

datatype pred =
  the_may
  | the_must
  | the_kill
  | the_gen
  | the_init
  | the_anadom

datatype var =
  the_u

abbreviation "may == PosLit the_may"
abbreviation "must == PosLit the_must"
abbreviation NegLit_BV ("¬may") **where**
  "¬may ≡ NegLit the_may"
abbreviation "kill == PosLit the_kill"
abbreviation NegLit_kill ("¬kill") **where**
  "¬kill ≡ NegLit the_kill"
abbreviation "gen == PosLit the_gen"
abbreviation "init == PosLit the_init"
abbreviation "anadom == PosLit the_anadom"

fun s_BV :: "pred ⇒ nat" **where**
  "s_BV the_kill = 0"
| "s_BV the_gen = 0"
| "s_BV the_init = 0"
| "s_BV the_anadom = 0"
| "s_BV the_may = 1"
| "s_BV the_must = 2"

datatype ('n,'a,'d) cst =
  is_Node: Node (the_Node: 'n)
  | is_Elem: Elem (the_Elem: 'd)
  | is_Action: Action (the_Action: "'a")

abbreviation may_Cls
  :: "(var, ('n,'v,'d) cst) id list ⇒
      (pred, var, ('n,'v,'d) cst) rh list ⇒
      (pred, var, ('n,'v,'d) cst) clause" ("may⟨_⟩ :- _ .") **where**
   "may⟨ids⟩ :- ls. ≡ Cls the_may ids ls"

```
abbreviation must_Cls
  :: "(var, ('n,'v,'d) cst) id list ⇒
      (pred, var, ('n,'v,'d) cst) rh list ⇒
      (pred, var, ('n,'v,'d) cst) clause" ("must⟨_⟩ :- _ .") where
  "must⟨ids⟩ :- ls. ≡ Cls the_must ids ls"

abbreviation init_Cls
  :: "(var, ('n,'v,'d) cst) id list ⇒
      (pred, var, ('n,'v,'d) cst) rh list ⇒
      (pred, var, ('n,'v,'d) cst) clause" ("init⟨_⟩ :- _ .") where
  "init⟨ids⟩ :- ls. ≡ Cls the_init ids ls"

abbreviation anadom_Cls
  :: "(var, ('n,'v,'d) cst) id list ⇒
      (pred, var, ('n,'v,'d) cst) rh list ⇒
      (pred, var, ('n,'v,'d) cst) clause" ("anadom⟨_⟩ :- _ .") where
  "anadom⟨ids⟩ :- ls. ≡ Cls the_anadom ids ls"

abbreviation kill_Cls
  :: "(var, ('n,'v,'d) cst) id list ⇒
      (pred, var, ('n,'v,'d) cst) rh list ⇒
      (pred, var, ('n,'v,'d) cst) clause" ("kill⟨_⟩ :- _ .") where
  "kill⟨ids⟩ :- ls. ≡ Cls the_kill ids ls"

abbreviation gen_Cls
  :: "(var, ('n,'v,'d) cst) id list ⇒
      (pred, var, ('n,'v,'d) cst) rh list ⇒
      (pred, var, ('n,'v,'d) cst) clause" ("gen⟨_⟩ :- _ .") where
  "gen⟨ids⟩ :- ls. ≡ Cls the_gen ids ls"

abbreviation BV_lh :: "(var, ('n,'v,'d) cst) id list ⇒ (pred, var, ('n,'v,'d) cst) lh"
  ("may⟨_⟩.") where
  "may⟨ids⟩. ≡ (the_may, ids)"

abbreviation must_lh :: "(var, ('n,'v,'d) cst) id list ⇒ (pred, var, ('n,'v,'d) cst) lh"
  ("must⟨_⟩.") where
  "must⟨ids⟩. ≡ (the_must, ids)"

abbreviation init_lh :: "(var, ('n,'v,'d) cst) id list ⇒ (pred, var, ('n,'v,'d) cst) lh"
  ("init⟨_⟩.") where
  "init⟨ids⟩. ≡ (the_init, ids)"

abbreviation dom_lh :: "(var, ('n,'v,'d) cst) id list ⇒ (pred, var, ('n,'v,'d) cst) lh"
  ("anadom⟨_⟩.") where
  "anadom⟨ids⟩. ≡ (the_anadom, ids)"

abbreviation u :: "(var, 'a) id" where
  "u == Var the_u"

abbreviation Cst_N :: "'n ⇒ (var, ('n, 'a, 'd) cst) id" where
  "Cst_N q == Cst (Node q)"

abbreviation Cst_E :: "'d ⇒ (var, ('n, 'a, 'd) cst) id" where
  "Cst_E e == Cst (Elem e)"

abbreviation Cst_A :: "'a ⇒ (var, ('n, 'a, 'd) cst) id" where
  "Cst_A α == Cst (Action α)"

abbreviation the_Node_id :: "(var, ('n, 'a, 'd) cst) id ⇒ 'n" where
  "the_Node_id ident == the_Node (the_Cst ident)"

abbreviation the_Elem_id :: "(var, ('n, 'a, 'd) cst) id ⇒ 'd" where
```

```
  "the_Elem_id ident == the_Elem (the_Cst ident)"

abbreviation the_Action_id :: "(var, ('n, 'a, 'd) cst) id ⇒ 'a" where
  "the_Action_id ident == the_Action (the_Cst ident)"

abbreviation is_Elem_id :: "(var, ('n, 'a, 'd) cst) id ⇒ bool" where
  "is_Elem_id ident == is_Cst ident ∧ is_Elem (the_Cst ident)"

abbreviation is_Node_id :: "(var, ('n, 'a, 'd) cst) id ⇒ bool" where
  "is_Node_id ident == is_Cst ident ∧ is_Node (the_Cst ident)"

abbreviation is_Action_id :: "(var, ('n, 'a, 'd) cst) id ⇒ bool" where
  "is_Action_id ident == is_Cst ident ∧ is_Action (the_Cst ident)"
```

11.2 Forward may-analysis

```
locale analysis_BV_forward_may = finite_program_graph pg
  for pg :: "('n::finite,'a) program_graph" +
  fixes analysis_dom :: "'d set"
  fixes kill_set :: "('n,'a) edge ⇒ 'd set"
  fixes gen_set :: "('n,'a) edge ⇒ 'd set"
  fixes d_init :: "'d set"
  assumes "finite analysis_dom"
  assumes "d_init ⊆ analysis_dom"
  assumes "∀e. gen_set e ⊆ analysis_dom"
  assumes "∀e. kill_set e ⊆ analysis_dom"
begin

lemma finite_d_init: "finite d_init"
  ⟨proof⟩

interpretation LTS edges ⟨proof⟩

definition "S_hat" :: "('n,'a) edge ⇒ 'd set ⇒ 'd set" ("S^_E⟦_⟧ _") where
  "S^_E⟦e⟧ R = (R - kill_set e) ∪ gen_set e"

lemma S_hat_mono:
  assumes "d1 ⊆ d2"
  shows "S^_E⟦e⟧ d1 ⊆ S^_E⟦e⟧ d2"
  ⟨proof⟩

fun S_hat_edge_list :: "('n,'a) edge list ⇒ 'd set ⇒ 'd set" ("S^_Es⟦_⟧ _") where
  "S^_Es⟦[]⟧ R = R" |
  "S^_Es⟦e # π⟧ R = S^_Es⟦π⟧ (S^_E⟦e⟧ R)"

lemma S_hat_edge_list_def2:
  "S^_Es⟦π⟧ R = foldl (λa b. S^_E⟦b⟧ a) R π"
⟨proof⟩

lemma S_hat_edge_list_append[simp]:
  "S^_Es⟦xs @ ys⟧ R = S^_Es⟦ys⟧ (S^_Es⟦xs⟧ R)"
  ⟨proof⟩

lemma S_hat_edge_list_mono:
  assumes "R1 ⊆ R2"
  shows "S^_Es⟦π⟧ R1 ⊆ S^_Es⟦π⟧ R2"
⟨proof⟩

definition S_hat_path :: "('n list × 'a list) ⇒ 'd set ⇒ 'd set" ("S^_P⟦_⟧ _") where
  "S^_P⟦π⟧ R = S^_Es⟦LTS.transition_list π⟧ R"

lemma S_hat_path_mono:
  assumes "R1 ⊆ R2"
  shows "S^_P⟦π⟧ R1 ⊆ S^_P⟦π⟧ R2"
```

⟨*proof*⟩

fun ana_kill_edge_d :: "('n, 'a) edge ⇒ 'd ⇒ (pred, var, ('n, 'a, 'd) cst) clause" **where**
"ana_kill_edge_d (q_o, α, q_s) d = kill⟨[Cst_N q_o, Cst_A α, Cst_N q_s, Cst_E d]⟩ :- []."

definition ana_kill_edge :: "('n, 'a) edge ⇒ (pred, var, ('n, 'a, 'd) cst) clause set" **where**
"ana_kill_edge e = ana_kill_edge_d e ‘ (kill_set e)"

lemma kill_set_eq_kill_set_inter_analysis_dom: "kill_set e = kill_set e ∩ analysis_dom"
⟨*proof*⟩

fun ana_gen_edge_d :: "('n, 'a) edge ⇒ 'd ⇒ (pred, var, ('n, 'a, 'd) cst) clause" **where**
"ana_gen_edge_d (q_o, α, q_s) d = gen⟨[Cst_N q_o, Cst_A α, Cst_N q_s, Cst_E d]⟩ :- []."

definition ana_gen_edge :: "('n, 'a) edge ⇒ (pred, var, ('n, 'a, 'd) cst) clause set" **where**
"ana_gen_edge e = ana_gen_edge_d e ‘ (gen_set e)"

lemma gen_set_eq_gen_set_inter_analysis_dom: "gen_set e = gen_set e ∩ analysis_dom"
⟨*proof*⟩

definition ana_init :: "'d ⇒ (pred, var, ('n, 'a, 'd) cst) clause" **where**
"ana_init d = init⟨[Cst_E d]⟩ :- []."

definition ana_anadom :: "'d ⇒ (pred, var, ('n, 'a, 'd) cst) clause" **where**
"ana_anadom d = anadom⟨[Cst_E d]⟩ :- []."

definition ana_entry_node :: "(pred, var, ('n, 'a, 'd) cst) clause" **where**
"ana_entry_node = may⟨[Cst_N start,u]⟩ :- [init[u]]."

fun ana_edge :: "('n, 'a) edge ⇒ (pred, var, ('n, 'a, 'd) cst) clause set" **where**
"ana_edge (q_o, α, q_s) =
{
may⟨[Cst_N q_s, u]⟩ :-
[
may[Cst_N q_o, u],
¬kill[Cst_N q_o, Cst_A α, Cst_N q_s, u]
].
,
may⟨[Cst_N q_s, u]⟩ :- [gen[Cst_N q_o, Cst_A α, Cst_N q_s, u]].
}"

definition ana_must :: "'n ⇒ (pred, var, ('n, 'a, 'd) cst) clause" **where**
"ana_must q = must⟨[Cst_N q,u]⟩ :- [¬may[Cst_N q,u], anadom[u]]."

lemma ana_must_meta_var:
assumes "ϱ ⊨$_{cls}$ must⟨[Cst_N q,u]⟩ :- [¬may[Cst_N q,u], anadom[u]]."
shows "ϱ ⊨$_{cls}$ must⟨[Cst_N q,v]⟩ :- [¬may[Cst_N q,v], anadom[v]]."
⟨*proof*⟩

definition ana_pg_fw_may :: "(pred, var, ('n, 'a, 'd) cst) clause set" **where**
"ana_pg_fw_may = ⋃(ana_edge ‘ edges)
∪ ana_init ‘ d_init
∪ ana_anadom ‘ analysis_dom
∪ ⋃(ana_kill_edge ‘ edges)
∪ ⋃(ana_gen_edge ‘ edges)
∪ ana_must ‘ UNIV
∪ {ana_entry_node}"

lemma ana_entry_node_meta_var:
assumes "ϱ ⊨$_{cls}$ may⟨[Cst_N start,u]⟩ :- [init[u]]."
shows "ϱ ⊨$_{cls}$ may⟨[Cst_N start,u]⟩ :- [init[u]]."
⟨*proof*⟩

definition summarizes_fw_may :: "(pred, ('n, 'a, 'd) cst) pred_val ⇒ bool" **where**
"summarizes_fw_may ϱ ⟷
(∀ π d. π ∈ path_with_word_from start ⟶ d ∈ S^$_P$⟦π⟧ d_init ⟶
ϱ ⊨$_{lh}$ (may⟨[Cst$_N$ (LTS.end_of π), Cst$_E$ d]⟩.))"

lemma S_hat_path_append:
assumes "length qs = length w"
shows "S^$_P$⟦(qs @ [qnminus1, qn], w @ [α])⟧ d_init =
S^$_E$⟦(qnminus1, α, qn)⟧ (S^$_P$⟦(qs @ [qnminus1], w)⟧ d_init)"
⟨*proof*⟩

lemma ana_pg_fw_may_stratified: "strat_wf s_BV ana_pg_fw_may"
⟨*proof*⟩

lemma finite_ana_edge_edgeset: "finite (⋃ (ana_edge ‘ edges))"
⟨*proof*⟩

lemma finite_ana_kill_edgeset: "finite (⋃ (ana_kill_edge ‘ edges))"
⟨*proof*⟩

lemma finite_ana_gen_edgeset: "finite (⋃ (ana_gen_edge ‘ edges))"
⟨*proof*⟩

lemma finite_ana_anadom_edgeset: "finite (ana_anadom ‘ analysis_dom)"
⟨*proof*⟩

lemma ana_pg_fw_may_finite: "finite ana_pg_fw_may"
⟨*proof*⟩

fun vars_lh :: "('p,'x,'e) lh ⇒ 'x set" **where**
"vars_lh (p,ids) = vars_ids ids"

lemma not_kill:
fixes ϱ :: "(pred, ('n, 'a, 'd) cst) pred_val"
assumes "d ∉ kill_set(q$_o$, α, q$_s$)"
assumes "ϱ ⊨$_{lst}$ ana_pg_fw_may s_BV"
shows "ϱ ⊨$_{rh}$ ¬kill[Cst$_N$ q$_o$, Cst$_A$ α, Cst$_N$ q$_s$, Cst$_E$ d]"
⟨*proof*⟩

lemma S_hat_edge_list_subset_analysis_dom:
assumes "d ∈ S^$_{Es}$⟦π⟧ d_init"
shows "d ∈ analysis_dom"
⟨*proof*⟩

lemma S_hat_path_subset_analysis_dom:
assumes "d ∈ S^$_P$⟦(ss,w)⟧ d_init"
shows "d ∈ analysis_dom"
⟨*proof*⟩

lemma S_hat_path_last:
assumes "length qs = length w"
shows "S^$_P$⟦(qs @ [qnminus1, qn], w @ [α])⟧ d_init =
S^$_E$⟦(qnminus1, α, qn)⟧ (S^$_P$⟦(qs @ [qnminus1], w)⟧ d_init)"
⟨*proof*⟩

lemma gen_sound:
assumes "d ∈ gen_set (q, α, q')"
assumes "(q, α, q') ∈ edges"
assumes "ϱ ⊨$_{lst}$ ana_pg_fw_may s_BV"
shows "ϱ ⊨$_{cls}$ gen⟨[Cst$_N$ q, Cst$_A$ α, Cst$_N$ q', Cst$_E$ d]⟩ :- [] ."
⟨*proof*⟩

lemma sound_ana_pg_fw_may':

assumes "(ss,w) ∈ path_with_word_from start"
assumes "d ∈ S^$_P$⟦(ss,w)⟧ d_init"
assumes "ϱ ⊨$_{lst}$ ana_pg_fw_may s_BV"
shows "ϱ ⊨$_{lh}$ may⟨[Cst$_N$ (LTS.end_of (ss, w)), Cst$_E$ d]⟩."
⟨*proof*⟩

theorem sound_ana_pg_fw_may:
assumes "ϱ ⊨$_{lst}$ ana_pg_fw_may s_BV"
shows "summarizes_fw_may ϱ"
⟨*proof*⟩

end

11.3 Backward may-analysis

locale analysis_BV_backward_may = finite_program_graph pg
for pg :: "('n::finite,'a) program_graph" +
fixes analysis_dom :: "'d set"
fixes kill_set :: "('n,'a) edge ⇒ 'd set"
fixes gen_set :: "('n,'a) edge ⇒ 'd set"
fixes d_init :: "'d set"
assumes "finite edges"
assumes "finite analysis_dom"
assumes "d_init ⊆ analysis_dom"
assumes "∀e. gen_set e ⊆ analysis_dom"
assumes "∀e. kill_set e ⊆ analysis_dom"
begin

interpretation LTS edges ⟨*proof*⟩

definition S_hat :: "('n,'a) edge ⇒ 'd set ⇒ 'd set" ("S^$_E$⟦_⟧ _") **where**
"S^$_E$⟦e⟧ R = (R - kill_set e) ∪ gen_set e"

lemma S_hat_mono:
assumes "R1 ⊆ R2"
shows "S^$_E$⟦e⟧ R1 ⊆ S^$_E$⟦e⟧ R2"
⟨*proof*⟩

fun S_hat_edge_list :: "('n,'a) edge list ⇒ 'd set ⇒ 'd set" ("S^$_{Es}$⟦_⟧ _") **where**
"S^$_{Es}$⟦[]⟧ R = R" |
"S^$_{Es}$⟦(e # π)⟧ R = S^$_E$⟦e⟧ (S^$_{Es}$⟦π⟧ R)"

lemma S_hat_edge_list_def2:
"S^$_{Es}$⟦π⟧ R = foldr S_hat π R"
⟨*proof*⟩

lemma S_hat_edge_list_append[simp]:
"S^$_{Es}$⟦(xs @ ys)⟧ R = S^$_{Es}$⟦xs⟧ (S^$_{Es}$⟦ys⟧ R)"
⟨*proof*⟩

lemma S_hat_edge_list_mono:
assumes "d1 ⊆ d2"
shows "S^$_{Es}$⟦π⟧ d1 ⊆ S^$_{Es}$⟦π⟧ d2"
⟨*proof*⟩

definition S_hat_path :: "('n list × 'a list) ⇒ 'd set ⇒ 'd set" ("S^$_P$⟦_⟧ _") **where**
"S^$_P$⟦π⟧ R = S^$_{Es}$⟦(transition_list π)⟧ R"

definition summarizes_bw_may :: "(pred, ('n, 'a, 'd) cst) pred_val ⇒ bool" **where**
"summarizes_bw_may ϱ ⟷ (∀π d. π ∈ path_with_word_to end ⟶ d ∈ S^$_P$⟦π⟧ d_init ⟶
ϱ ⊨$_{lh}$ may⟨[Cst$_N$ (start_of π), Cst$_E$ d]⟩.)"

lemma kill_subs_analysis_dom: "(kill_set (rev_edge e)) ⊆ analysis_dom"
⟨*proof*⟩

lemma gen_subs_analysis_dom: "(gen_set (rev_edge e)) ⊆ analysis_dom"
⟨*proof*⟩

interpretation fw_may: analysis_BV_forward_may
pg_rev analysis_dom "λe. (kill_set (rev_edge e))" "(λe. gen_set (rev_edge e))" d_init
⟨*proof*⟩

abbreviation ana_pg_bw_may **where**
"ana_pg_bw_may == fw_may.ana_pg_fw_may"

lemma rev_end_is_start:
assumes "ss ≠ []"
assumes "end_of (ss, w) = end"
shows "start_of (rev ss, rev w) = fw_may.start"
⟨*proof*⟩

lemma S_hat_edge_list_forward_backward:
"S^$_{Es}$⟦ss⟧ d_init = fw_may.S_hat_edge_list (rev_edge_list ss) d_init"
⟨*proof*⟩

lemma S_hat_path_forward_backward:
assumes "(ss,w) ∈ path_with_word"
shows "S^$_P$⟦(ss, w)⟧ d_init = fw_may.S_hat_path (rev ss, rev w) d_init"
⟨*proof*⟩

lemma summarizes_bw_may_forward_backward':
assumes "(ss,w) ∈ path_with_word"
assumes "end_of (ss,w) = end"
assumes "d ∈ S^$_P$⟦(ss,w)⟧ d_init"
assumes "fw_may.summarizes_fw_may ϱ"
shows "ϱ ⊨$_{lh}$ may⟨[Cst$_N$ (start_of (ss, w)), Cst$_E$ d]⟩."
⟨*proof*⟩

lemma summarizes_dl_BV_forward_backward:
assumes "fw_may.summarizes_fw_may ϱ"
shows "summarizes_bw_may ϱ"
⟨*proof*⟩

theorem sound_ana_pg_bw_may:
assumes "ϱ ⊨$_{lst}$ ana_pg_bw_may s_BV"
shows "summarizes_bw_may ϱ"
⟨*proof*⟩

end

11.4 Forward must-analysis

locale analysis_BV_forward_must = finite_program_graph pg
for pg :: "('n::finite,'a) program_graph" +
fixes analysis_dom :: "'d set"
fixes kill_set :: "('n,'a) edge ⇒ 'd set"
fixes gen_set :: "('n,'a) edge ⇒ 'd set"
fixes d_init :: "'d set"
assumes "finite analysis_dom"
assumes "d_init ⊆ analysis_dom"
begin

lemma finite_d_init: "finite d_init"
⟨*proof*⟩

interpretation LTS edges ⟨*proof*⟩

definition S_hat :: "('n,'a) edge ⇒ 'd set ⇒ 'd set" ("S^$_E$⟦_⟧ _") **where**
"S^$_E$⟦e⟧ R = (R - kill_set e) ∪ gen_set e"

lemma S_hat_mono:
assumes "R1 ⊆ R2"
shows "S^$_E$⟦e⟧ R1 ⊆ S^$_E$⟦e⟧ R2"
⟨*proof*⟩

fun S_hat_edge_list :: "('n,'a) edge list ⇒ 'd set ⇒ 'd set" ("S^$_{Es}$⟦_⟧ _") **where**
"S^$_{Es}$⟦[]⟧ R = R" |
"S^$_{Es}$⟦(e # π)⟧ R = S^$_{Es}$⟦π⟧ (S^$_E$⟦e⟧ R)"

lemma S_hat_edge_list_def2:
"S^$_{Es}$⟦π⟧ R = foldl (λa b. S^$_E$⟦b⟧ a) R π"
⟨*proof*⟩

lemma S_hat_edge_list_append[simp]:
"S^$_{Es}$⟦(xs @ ys)⟧ R = S^$_{Es}$⟦ys⟧ (S^$_{Es}$⟦xs⟧ R)"
⟨*proof*⟩

lemma S_hat_edge_list_mono:
assumes "R1 ⊆ R2"
shows "S^$_{Es}$⟦π⟧ R1 ⊆ S^$_{Es}$⟦π⟧ R2"
⟨*proof*⟩

definition S_hat_path :: "('n list × 'a list) ⇒ 'd set ⇒ 'd set" ("S^$_P$⟦_⟧ _") **where**
"S^$_P$⟦π⟧ R = S^$_{Es}$⟦LTS.transition_list π⟧ R"

lemma S_hat_path_mono:
assumes "R1 ⊆ R2"
shows "S^$_P$⟦π⟧ R1 ⊆ S^$_P$⟦π⟧ R2"
⟨*proof*⟩

definition summarizes_fw_must :: "(pred, ('n, 'a, 'd) cst) pred_val ⇒ bool" **where**
"summarizes_fw_must ϱ ⟷
(∀q d.
ϱ ⊨$_{lh}$ must⟨[q, d]⟩. ⟶
(∀π. π ∈ path_with_word ⟶
start_of π = start ⟶
end_of π = the_Node$_{id}$ q ⟶
(the_Elem$_{id}$ d) ∈ S^$_P$⟦π⟧ d_init))"

interpretation fw_may: analysis_BV_forward_may
pg analysis_dom "λe. analysis_dom - (kill_set e)" "(λe. analysis_dom - gen_set e)"
"analysis_dom - d_init"
⟨*proof*⟩

abbreviation ana_pg_fw_must **where**
"ana_pg_fw_must == fw_may.ana_pg_fw_may"

lemma opposite_lemma:
assumes "d ∈ analysis_dom"
assumes "¬d ∈ fw_may.S_hat_edge_list π (analysis_dom - d_init)"
shows "d ∈ S^$_{Es}$⟦π⟧ d_init"
⟨*proof*⟩

lemma opposite_lemma_path:
assumes "d ∈ analysis_dom"
assumes "¬d ∈ fw_may.S_hat_path π (analysis_dom - d_init)"
shows "d ∈ S^$_P$⟦π⟧ d_init"
⟨*proof*⟩

lemma the_must_only_ana_must: "the_must ∉ preds_dl (ana_pg_fw_must - (fw_may.ana_must ‘ UNIV))"
⟨*proof*⟩

lemma agree_off_rh:
assumes "∀p. p ≠ p' ⟶ ϱ' p = ϱ p"
assumes "p' ∉ preds_rh rh"
shows "⟦rh⟧$_{rh}$ ϱ' σ ⟷ ⟦rh⟧$_{rh}$ ϱ σ"
⟨*proof*⟩

definition preds_rhs **where**
"preds_rhs rhs = ⋃(preds_rh ‘ set rhs)"

lemma preds_cls_preds_rhs: "preds_cls (Cls p ids rhs) = {p} ∪ preds_rhs rhs"
⟨*proof*⟩

lemma agree_off_rhs:
assumes "∀p. p ≠ p' ⟶ ϱ' p = ϱ p"
assumes "p' ∉ preds_rhs rhs"
shows "⟦rhs⟧$_{rhs}$ ϱ' σ ⟷ ⟦rhs⟧$_{rhs}$ ϱ σ"
⟨*proof*⟩

lemma agree_off_lh:
assumes "∀p. p ≠ p' ⟶ ϱ' p = ϱ p"
assumes "p' ∉ preds_lh lh"
shows "⟦lh⟧$_{lh}$ ϱ' σ ⟷ ⟦lh⟧$_{lh}$ ϱ σ"
⟨*proof*⟩

lemma agree_off_cls:
assumes "∀p. p ≠ p' ⟶ ϱ' p = ϱ p"
assumes "p' ∉ preds_cls c"
shows "⟦c⟧$_{cls}$ ϱ' σ ⟷ ⟦c⟧$_{cls}$ ϱ σ"
⟨*proof*⟩

lemma agree_off_solves_cls:
assumes "∀p. p ≠ p' ⟶ ϱ' p = ϱ p"
assumes "p' ∉ preds_cls c"
shows "ϱ' ⊨$_{cls}$ c ⟷ ϱ ⊨$_{cls}$ c"
⟨*proof*⟩

lemma agree_off_dl:
assumes "∀p. p ≠ p' ⟶ ϱ' p = ϱ p"
assumes "p' ∉ preds_dl dl"
shows "ϱ' ⊨$_{dl}$ dl ⟷ ϱ ⊨$_{dl}$ dl"
⟨*proof*⟩

lemma is_Cst_if_init:
assumes "ϱ ⊨$_{lst}$ ana_pg_fw_must s_BV"
assumes "ϱ ⊨$_{lh}$ init⟨[d]⟩."
shows "is_Cst d"
⟨*proof*⟩

lemma is_Cst_if_anadom:
assumes "ϱ ⊨$_{lst}$ ana_pg_fw_must s_BV"
assumes "ϱ ⊨$_{lh}$ anadom⟨[d]⟩."
shows "is_Cst d"
⟨*proof*⟩

lemma if_init:
assumes "ϱ ⊨$_{lst}$ ana_pg_fw_must s_BV"
assumes "ϱ ⊨$_{lh}$ init⟨[d]⟩."
shows "is_Elem$_{id}$ d ∧ the_Elem$_{id}$ d ∈ (analysis_dom - d_init)"
⟨*proof*⟩

lemma if_anadom:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
assumes "$\varrho \models_{lh}$ anadom⟨[d]⟩."
shows "is_Elem$_{id}$ d ∧ the_Elem$_{id}$ d ∈ analysis_dom"
⟨*proof*⟩

lemma is_elem_if_init:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
assumes "$\varrho \models_{lh}$ init⟨[Cst d]⟩."
shows "is_Elem d"
⟨*proof*⟩

lemma not_init_node:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
shows "$\neg\varrho \models_{lh}$ init⟨[Cst$_N$ q]⟩."
⟨*proof*⟩

lemma not_init_action:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
shows "$\neg\varrho \models_{lh}$ init⟨[Cst$_A$ q]⟩."
⟨*proof*⟩

lemma in_analysis_dom_if_init':
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
assumes "$\varrho \models_{lh}$ init⟨[Cst$_E$ d]⟩."
shows "d ∈ analysis_dom"
⟨*proof*⟩

lemma in_analysis_dom_if_init:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
assumes "$\varrho \models_{lh}$ init⟨[d]⟩."
shows "the_Elem$_{id}$ d ∈ analysis_dom"
⟨*proof*⟩

lemma not_anadom_node:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
shows "$\neg\varrho \models_{lh}$ anadom⟨[Cst$_N$ q]⟩."
⟨*proof*⟩

lemma not_anadom_action:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
shows "$\neg\varrho \models_{lh}$ anadom⟨[Cst$_A$ q]⟩."
⟨*proof*⟩

lemma in_analysis_dom_if_anadom':
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
assumes "$\varrho \models_{lh}$ anadom⟨[Cst$_E$ d]⟩."
shows "d ∈ analysis_dom"
⟨*proof*⟩

lemma in_analysis_dom_if_anadom:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
assumes "$\varrho \models_{lh}$ anadom⟨[d]⟩."
shows "the_Elem$_{id}$ d ∈ analysis_dom ∧ is_Elem$_{id}$ d"
⟨*proof*⟩

lemma must_fst_id_is_Cst:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
assumes "$\varrho \models_{lh}$ must⟨[q,d]⟩."
shows "is_Cst q"
⟨*proof*⟩

lemma must_snd_id_is_Cst:

assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
assumes "$\varrho \models_{lh}$ must⟨[q,d]⟩."
shows "is_Cst d"
⟨*proof*⟩

lemma if_must:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
assumes "$\varrho \models_{lh}$ must⟨[q,d]⟩."
shows
"$\varrho \models_{rh}$ ¬may[q, d] ∧ $\varrho \models_{lh}$ anadom⟨[d]⟩. ∧ is_Node$_{id}$ q ∧is_Elem$_{id}$ d ∧ the_Elem$_{id}$ d ∈ analysis_dom"
⟨*proof*⟩

lemma not_must_and_may:
assumes "[Node q, Elem d] ∈ ϱ the_must"
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
assumes "[Node q, Elem d] ∈ ϱ the_may"
shows False
⟨*proof*⟩

lemma not_solves_must_and_may:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
assumes "$\varrho \models_{lh}$ must⟨[Cst$_N$ q, Cst$_E$ d]⟩."
assumes "$\varrho \models_{lh}$ may⟨[Cst$_N$ q, Cst$_E$ d]⟩."
shows "False"
⟨*proof*⟩

lemma anadom_if_must:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
assumes "$\varrho \models_{lh}$ must⟨[q, d]⟩."
shows "$\varrho \models_{lh}$ anadom⟨[d]⟩."
⟨*proof*⟩

lemma not_must_action:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
shows "¬$\varrho \models_{lh}$ must⟨[Cst$_A$ q,d]⟩."
⟨*proof*⟩

lemma is_encode_elem_if_must_right_arg:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
assumes "$\varrho \models_{lh}$ must⟨[q, d]⟩."
shows "∃d'. d = Cst$_E$ d'"
⟨*proof*⟩

lemma not_must_element:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
shows "¬$\varrho \models_{lh}$ must⟨[Cst$_E$ q,d]⟩."
⟨*proof*⟩

lemma is_encode_node_if_must_left_arg:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
assumes "$\varrho \models_{lh}$ must⟨[q, d]⟩."
shows "∃q'. q = Cst$_N$ q'"
⟨*proof*⟩

lemma in_analysis_dom_if_must:
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
assumes "$\varrho \models_{lh}$ must⟨[q, d]⟩."
shows "the_Elem$_{id}$ d ∈ analysis_dom"
⟨*proof*⟩

lemma sound_ana_pg_fw_must':
assumes "$\varrho \models_{lst}$ ana_pg_fw_must s_BV"
assumes "$\varrho \models_{lh}$ must⟨[q, d]⟩."

assumes "π ∈ path_with_word_from_to start (the_Node$_{id}$ q)"
shows "the_Elem$_{id}$ d ∈ S^$_P$⟦π⟧ d_init"
⟨*proof*⟩

theorem sound_ana_pg_fw_must:
assumes "ϱ ⊨$_{lst}$ ana_pg_fw_must s_BV"
shows "summarizes_fw_must ϱ"
⟨*proof*⟩

end

11.5 Backward must-analysis

locale analysis_BV_backward_must = finite_program_graph pg
for pg :: "('n::finite,'a) program_graph" +
fixes analysis_dom :: "'d set"
fixes kill_set :: "('n,'a) edge ⇒ 'd set"
fixes gen_set :: "('n,'a) edge ⇒ 'd set"
fixes d_init :: "'d set"
assumes "finite analysis_dom"
assumes "d_init ⊆ analysis_dom"
begin

lemma finite_d_init: "finite d_init"
⟨*proof*⟩

interpretation LTS edges ⟨*proof*⟩

definition S_hat :: "('n,'a) edge ⇒ 'd set ⇒ 'd set" ("S^$_E$⟦_⟧ _") **where**
"S^$_E$⟦e⟧ R = (R - kill_set e) ∪ gen_set e"

lemma S_hat_mono:
assumes "R1 ⊆ R2"
shows "S^$_E$⟦e⟧ R1 ⊆ S^$_E$⟦e⟧ R2"
⟨*proof*⟩

fun S_hat_edge_list :: "('n,'a) edge list ⇒ 'd set ⇒ 'd set" ("S^$_{Es}$⟦_⟧ _") **where**
"S^$_{Es}$⟦[]⟧ R = R" |
"S^$_{Es}$⟦(e # π)⟧ R = S^$_E$⟦e⟧ (S^$_{Es}$⟦π⟧ R)"

lemma S_hat_edge_list_def2:
"S^$_{Es}$⟦π⟧ R = foldr S_hat π R"
⟨*proof*⟩

lemma S_hat_edge_list_append[simp]:
"S^$_{Es}$⟦xs @ ys⟧ R = S^$_{Es}$⟦xs⟧ (S^$_{Es}$⟦ys⟧ R)"
⟨*proof*⟩

lemma S_hat_edge_list_mono:
assumes "R1 ⊆ R2"
shows "S^$_{Es}$⟦π⟧ R1 ⊆ S^$_{Es}$⟦π⟧ R2"
⟨*proof*⟩

definition S_hat_path :: "('n list × 'a list) ⇒ 'd set ⇒ 'd set" ("S^$_P$⟦_⟧ _") **where**
"S^$_P$⟦π⟧ R = S^$_{Es}$⟦LTS.transition_list π⟧ R"

definition summarizes_bw_must :: "(pred, ('n, 'v, 'd) cst) pred_val ⇒ bool" **where**
"summarizes_bw_must ϱ ⟷
(∀q d.
ϱ ⊨$_{lh}$ must⟨[q, d]⟩. ⟶
(∀π. π ∈ path_with_word_from_to (the_Node$_{id}$ q) end ⟶ the_Elem$_{id}$ d ∈ S^$_P$⟦π⟧ d_init))"

interpretation fw_must: analysis_BV_forward_must
pg_rev analysis_dom "λe. (kill_set (rev_edge e))" "(λe. gen_set (rev_edge e))" d_init

⟨*proof*⟩

abbreviation ana_pg_bw_must **where**
"ana_pg_bw_must == fw_must.ana_pg_fw_must"

lemma rev_end_is_start:
assumes "ss ≠ []"
assumes "end_of (ss, w) = end"
shows "start_of (rev ss, rev w) = fw_must.start"
⟨*proof*⟩

lemma S_hat_edge_list_forward_backward:
"S^$_{Es}$⟦ss⟧ d_init = fw_must.S_hat_edge_list (rev_edge_list ss) d_init"
⟨*proof*⟩

lemma S_hat_path_forward_backward:
assumes "(ss,w) ∈ path_with_word"
shows "S^$_P$⟦(ss, w)⟧ d_init = fw_must.S_hat_path (rev ss, rev w) d_init"
⟨*proof*⟩

lemma summarizes_fw_must_forward_backward':
assumes "fw_must.summarizes_fw_must ϱ"
assumes "ϱ ⊨$_{lh}$ must⟨[q, d]⟩."
assumes "π ∈ path_with_word_from_to (the_Node$_{id}$ q) end"
shows "the_Elem$_{id}$ d ∈ S^$_P$⟦π⟧ d_init"
⟨*proof*⟩

lemma summarizes_bw_must_forward_backward:
assumes "fw_must.summarizes_fw_must ϱ"
shows "summarizes_bw_must ϱ"
⟨*proof*⟩

theorem sound_ana_pg_bw_must:
assumes "ϱ ⊨$_{lst}$ ana_pg_bw_must s_BV"
shows "summarizes_bw_must ϱ"
⟨*proof*⟩

end

end
theory Reaching_Definitions **imports** Bit_Vector_Framework **begin**

— We encode the Reaching Definitions analysis into Datalog. First we define the analysis, then we encode the analysis directly into Datalog and prove the encoding correct. Hereafter we encode it into Datalog again, but this time using our Bit-Vector Framework locale. We also prove this encoding correct. This latter encoding is described in our SAC 2024 paper.

12 Reaching Definitions

type_synonym ('n,'v) def = "'v * 'n option * 'n"

type_synonym ('n,'v) analysis_assignment = "'n ⇒ ('n,'v) def set"

12.1 What is defined on a path

fun def_action :: "'v action ⇒ 'v set" **where**
"def_action (x ::= a) = {x}"
| "def_action (Bool b) = {}"
| "def_action Skip = {}"

abbreviation def_edge :: "('n,'v action) edge ⇒ 'v set" **where**

```
  "def_edge == λ(q1, α, q2). def_action α"

definition def_of :: "'v ⇒ ('n,'v action) edge ⇒ ('n,'v) def" where
  "def_of == (λx (q1, α, q2). (x, Some q1, q2))"

definition def_var :: "('n,'v action) edge list ⇒ 'v ⇒ 'n ⇒ ('n,'v) def" where
  "def_var π x start = (if (∃e ∈ set π. x ∈ def_edge e)
                        then (def_of x (last (filter (λe. x ∈ def_edge e) π)))
                        else (x, None, start))"

definition def_path :: "('n list × 'v action list) ⇒ 'n ⇒ ('n,'v) def set" where
  "def_path π start = ((λx. def_var (LTS.transition_list π) x start) ‘ UNIV)"
```

12.2 Reaching Definitions in Datalog

```
datatype ('n,'v) RD_elem =
  RD_Node 'n
  | RD_Var 'v
  | Questionmark

datatype RD_var =
  the_u
  | the_v
  | the_w

datatype RD_pred =
  the_RD
  | the_VAR

abbreviation Cst_RDN :: "'n ⇒ (RD_var, ('n, 'v) RD_elem) id" where
  "Cst_RDN q == Cst (RD_Node q)"

fun Cst_RDN_Q :: "'n option ⇒ (RD_var, ('n, 'v) RD_elem) id" where
  "Cst_RDN_Q (Some q) = Cst (RD_Node q)"
| "Cst_RDN_Q None = Cst Questionmark"

abbreviation Cst_RDV :: "'v ⇒ (RD_var, ('n, 'v) RD_elem) id" where
  "Cst_RDV v == Cst (RD_Var v)"

abbreviation RD_Cls :: "(RD_var, ('n, 'v) RD_elem) id list ⇒ (RD_pred, RD_var, ('n, 'v) RD_elem) rh
list ⇒ (RD_pred, RD_var, ('n, 'v) RD_elem) clause" ("RD⟨_⟩ :- _ .") where
  "RD⟨args⟩ :- ls. ≡ Cls the_RD args ls"

abbreviation VAR_Cls :: "'v ⇒ (RD_pred, RD_var, ('n, 'v) RD_elem) clause" ("VAR⟨_⟩ :-.") where
  "VAR⟨x⟩ :-. == Cls the_VAR [Cst_RDV x] []"

abbreviation RD_lh :: "(RD_var, ('n, 'v) RD_elem) id list ⇒ (RD_pred, RD_var, ('n, 'v) RD_elem) lh"
("RD⟨_⟩.") where
  "RD⟨args⟩. ≡ (the_RD, args)"

abbreviation VAR_lh :: "'v ⇒ (RD_pred, RD_var, ('n, 'v) RD_elem) lh" ("VAR⟨_⟩.") where
  "VAR⟨x⟩. ≡ (the_VAR, [Cst_RDV x])"


abbreviation "RD == PosLit the_RD"
abbreviation "VAR == PosLit the_VAR"

abbreviation u :: "(RD_var, 'aa) id" where
  "u == Var the_u"

abbreviation v :: "(RD_var, 'aa) id" where
  "v == Var the_v"

abbreviation w :: "(RD_var, 'aa) id" where
```

```
  "w == Var the_w"

fun ana_edge :: "('n, 'v action) edge ⇒ (RD_pred, RD_var, ('n,'v) RD_elem) clause set" where
  "ana_edge (q_o, x ::= a, q_s) =
     {
        RD⟨[Cst_RDN q_s, u, v, w]⟩ :-
          [
            RD[Cst_RDN q_o, u, v, w],
            u ≠ (Cst_RDV x)
          ].
        ,
        RD⟨[Cst_RDN q_s, Cst_RDV x, Cst_RDN q_o, Cst_RDN q_s]⟩ :- [].
     }"
| "ana_edge (q_o, Bool b, q_s) =
     {
       RD⟨[Cst_RDN q_s, u, v, w]⟩ :-
         [
           RD[Cst_RDN q_o, u, v, w]
         ].
     }"
| "ana_edge (q_o, Skip, q_s) =
     {
       RD⟨[Cst_RDN q_s, u, v, w]⟩ :-
         [
           RD[Cst_RDN q_o, u, v, w]
         ].
     }"

definition ana_entry_node :: "'n ⇒ (RD_pred, RD_var, ('n,'v) RD_elem) clause set" where
  "ana_entry_node start =
     {
       RD⟨[Cst_RDN start, u, Cst Questionmark, Cst_RDN start]⟩ :-
         [
           VAR[u]
         ].
     }"


fun ana_RD :: "('n, 'v action) program_graph ⇒ (RD_pred, RD_var, ('n,'v) RD_elem) clause set" where
  "ana_RD (es,start,end) = ⋃(ana_edge ‘ es) ∪ ana_entry_node start"

definition var_contraints :: "(RD_pred, RD_var, ('n,'v) RD_elem) clause set" where
  "var_contraints = VAR_Cls ‘ UNIV"

type_synonym ('n,'v) quadruple = "'n *'v * 'n option * 'n"

fun summarizes_RD :: "(RD_pred,('n,'v) RD_elem) pred_val ⇒ ('n,'v action) program_graph ⇒ bool" where
  "summarizes_RD ϱ (es, start, end) =
    (∀π x q1 q2.
       π ∈ LTS.path_with_word_from es start ⟶
       (x, q1, q2) ∈ def_path π start ⟶
       ϱ ⊨_lh RD⟨[Cst_RDN (LTS.end_of π), Cst_RDV x, Cst_RDN_Q q1, Cst_RDN q2]⟩.)"

lemma def_var_x: "fst (def_var ts x start) = x"
  ⟨proof⟩

lemma last_def_transition:
  assumes "length ss = length w"
  assumes "x ∈ def_action α"
  assumes "(x, q1, q2) ∈ def_path (ss @ [s, s'], w @ [α]) start"
  shows "Some s = q1 ∧ s' = q2"
⟨proof⟩
```

lemma not_last_def_transition:
  assumes "length ss = length w"
  assumes "x ∉ def_action α"
  assumes "(x, q1, q2) ∈ def_path (ss @ [s, s'], w @ [α]) start"
  shows "(x, q1, q2) ∈ def_path (ss @ [s], w) start"
⟨*proof*⟩

theorem RD_sound':
  assumes "(ss,w) ∈ LTS.path_with_word_from es start"
  assumes "(x,q1,q2) ∈ def_path (ss,w) start"
  assumes "ϱ ⊨$_{dl}$ (var_contraints ∪ ana_RD (es, start, end))"
  shows "ϱ ⊨$_{lh}$ RD⟨[Cst$_{RDN}$ (LTS.end_of (ss, w)), Cst$_{RDV}$ x, Cst$_{RDN}$_Q q1, Cst$_{RDN}$ q2]⟩."
  ⟨*proof*⟩

theorem RD_sound:
  assumes "ϱ ⊨$_{dl}$ (var_contraints ∪ ana_RD pg)"
  shows "summarizes_RD ϱ pg"
  ⟨*proof*⟩

12.3 Reaching Definitions as Bit-Vector Framework analysis

locale analysis_RD = finite_program_graph pg
  for pg :: "('n::finite,'v::finite action) program_graph" +
  assumes "finite edges"
begin

interpretation LTS edges ⟨*proof*⟩

definition analysis_dom_RD :: "('n,'v) def set" **where**
  "analysis_dom_RD = UNIV × UNIV × UNIV"

fun kill_set_RD :: "('n,'v action) edge ⇒ ('n,'v) def set" **where**
  "kill_set_RD (q$_o$, x ::= a, q$_s$) = {x} × UNIV × UNIV"
| "kill_set_RD (q$_o$, Bool b, q$_s$) = {}"
| "kill_set_RD (v, Skip, vc) = {}"

fun gen_set_RD :: "('n,'v action) edge ⇒ ('n,'v) def set" **where**
  "gen_set_RD (q$_o$, x ::= a, q$_s$) = {x} × {Some q$_o$} × {q$_s$}"
| "gen_set_RD (q$_o$, Bool b, q$_s$) = {}"
| "gen_set_RD (v, Skip, vc) = {} "

definition d_init_RD :: " ('n,'v) def set" **where**
  "d_init_RD = (UNIV × {None} × {start})"

lemma finite_analysis_dom_RD: "finite analysis_dom_RD"
  ⟨*proof*⟩

lemma d_init_RD_subset_analysis_dom_RD:
  "d_init_RD ⊆ analysis_dom_RD"
  ⟨*proof*⟩

lemma gen_RD_subset_analysis_dom: "gen_set_RD e ⊆ analysis_dom_RD"
  ⟨*proof*⟩

lemma kill_RD_subset_analysis_dom: "kill_set_RD e ⊆ analysis_dom_RD"
  ⟨*proof*⟩

interpretation fw_may: analysis_BV_forward_may pg analysis_dom_RD kill_set_RD gen_set_RD d_init_RD
  ⟨*proof*⟩

lemma def_var_def_edge_S_hat:
  assumes "def_var π x start ∈ R"
  assumes "x ∉ def_edge t"

shows "def_var π x start $\in$ fw_may.S_hat t R"
⟨*proof*⟩

lemma def_var_S_hat_edge_list: "(def_var π) x start $\in$ fw_may.S_hat_edge_list π d_init_RD"
⟨*proof*⟩

lemma last_overwrites:
"def_var (π @ [(q1, x ::= exp, q2)]) x start = (x, Some q1, q2)"
⟨*proof*⟩

lemma S_hat_edge_list_last: "fw_may.S_hat_edge_list (π @ [e]) d_init_RD = fw_may.S_hat e (fw_may.S_hat_edge_lis
π d_init_RD)"
⟨*proof*⟩

lemma def_var_if_S_hat:
assumes "(x,q1,q2) $\in$ fw_may.S_hat_edge_list π d_init_RD"
shows "(x,q1,q2) = (def_var π) x start"
⟨*proof*⟩

lemma def_var_UNIV_S_hat_edge_list: "(λx. def_var π x start) ‘ UNIV = fw_may.S_hat_edge_list π d_init_RD"
⟨*proof*⟩

lemma def_path_S_hat_path: "def_path π start = fw_may.S_hat_path π d_init_RD"
⟨*proof*⟩

definition summarizes_RD :: "(pred, (’n,’v action,(’n,’v) def) cst) pred_val $\Rightarrow$ bool" **where**
"summarizes_RD ϱ $\longleftrightarrow$ ($\forall\pi$ d. π $\in$ path_with_word_from start $\longrightarrow$ d $\in$ def_path π start $\longrightarrow$
ϱ $\models_{lh}$ may⟨[Cst$_N$ (end_of π), Cst$_E$ d]⟩.)"

theorem RD_sound:
assumes "ϱ $\models_{lst}$ fw_may.ana_pg_fw_may s_BV"
shows "summarizes_RD ϱ"
⟨*proof*⟩

end

end
theory Live_Variables **imports** Reaching_Definitions **begin**

— We encode the Live Variables analysis into Datalog. First we define the analysis, then we encode it into Datalog using our Bit-Vector Framework locale. We also prove the encoding correct.

13 Live Variables Analysis

fun use_action :: "’v action $\Rightarrow$ ’v set" **where**
"use_action (x ::= a) = fv_arith a"
| "use_action (Bool b) = fv_boolean b"
| "use_action Skip = {}"

fun use_edge :: "(’n,’v action) edge $\Rightarrow$ ’v set" **where**
"use_edge (q1, α, q2) = use_action α"

definition use_edge_list :: "(’n,’v action) edge list $\Rightarrow$ ’v $\Rightarrow$ bool" **where**
"use_edge_list π x = ($\exists\pi$1 π2 e. π = π1 @ [e] @ π2 $\wedge$
x $\in$ use_edge e $\wedge$
($\neg$($\exists$e’ $\in$ set π1. x $\in$ def_edge e’)))"

definition use_path :: "’n list $\times$ ’v action list $\Rightarrow$ ’v set" **where**
"use_path π = {x. use_edge_list (LTS.transition_list π) x}"

locale analysis_LV = finite_program_graph pg
for pg :: "(’n::finite,’v::finite action) program_graph"
begin

interpretation LTS edges ⟨*proof*⟩

definition analysis_dom_LV :: "'v set" **where**
  "analysis_dom_LV = UNIV"

fun kill_set_LV :: "('n,'v action) edge ⇒ 'v set" **where**
  "kill_set_LV (q_o, x ::= a, q_s) = {x}"
| "kill_set_LV (q_o, Bool b, q_s) = {}"
| "kill_set_LV (v, Skip, vc) = {}"

fun gen_set_LV :: "('n,'v action) edge ⇒ 'v set" **where**
  "gen_set_LV (q_o, x ::= a, q_s) = fv_arith a"
| "gen_set_LV (q_o, Bool b, q_s) = fv_boolean b"
| "gen_set_LV (v, Skip, vc) = {}"

definition d_init_LV :: "'v set" **where**
  "d_init_LV = {}"

interpretation bw_may: analysis_BV_backward_may pg analysis_dom_LV kill_set_LV gen_set_LV d_init_LV
  ⟨*proof*⟩

lemma use_edge_list_S_hat_edge_list:
  assumes "use_edge_list π x"
  shows "x ∈ bw_may.S_hat_edge_list π d_init_LV"
  ⟨*proof*⟩

lemma S_hat_edge_list_use_edge_list:
  assumes "x ∈ bw_may.S_hat_edge_list π d_init_LV"
  shows "use_edge_list π x"
  ⟨*proof*⟩

lemma use_edge_list_set_S_hat_edge_list:
  "{x. use_edge_list π x} = bw_may.S_hat_edge_list π d_init_LV"
  ⟨*proof*⟩

lemma use_path_S_hat_path: "use_path π = bw_may.S_hat_path π d_init_LV"
  ⟨*proof*⟩

definition summarizes_LV :: "(pred, ('n,'v action,'v) cst) pred_val ⇒ bool" **where**
  "summarizes_LV ϱ ⟷ (∀π d. π ∈ path_with_word_to end ⟶ d ∈ use_path π ⟶
                        ϱ $\models_{lh}$ may⟨[Cst_N (start_of π), Cst_E d]⟩.)"

theorem LV_sound:
  assumes "ϱ $\models_{lst}$ bw_may.ana_pg_bw_may s_BV"
  shows "summarizes_LV ϱ"
⟨*proof*⟩

end

end
theory Available_Expressions **imports** Reaching_Definitions **begin**

— We encode the Available Expressions analysis into Datalog. First we define the analysis, then we encode it into Datalog using our Bit-Vector Framework locale. We also prove the encoding correct.

14 Available Expressions

fun ae_arith :: "'v arith ⇒ 'v arith set" **where**
  "ae_arith (Integer i) = {}"
| "ae_arith (Arith_Var v) = {}"
| "ae_arith (Arith_Op a1 opr a2) = ae_arith a1 ∪ ae_arith a1 ∪ {Arith_Op a1 opr a2}"
| "ae_arith (Minus a) = ae_arith a"

lemma finite_ae_arith: "finite (ae_arith a)"
⟨*proof*⟩

fun ae_boolean :: "'v boolean ⇒ 'v arith set" **where**
"ae_boolean true = {}"
| "ae_boolean false = {}"
| "ae_boolean (Rel_Op a1 opr a2) = ae_arith a1 ∪ ae_arith a2"
| "ae_boolean (Bool_Op b1 opr b2) = ae_boolean b1 ∪ ae_boolean b2"
| "ae_boolean (Neg b) = ae_boolean b"

lemma finite_ae_boolean: "finite (ae_boolean b)"
⟨*proof*⟩

fun aexp_action :: "'v action ⇒ 'v arith set" **where**
"aexp_action (x ::= a) = ae_arith a"
| "aexp_action (Bool b) = ae_boolean b"
| "aexp_action Skip = {}"

lemma finite_aexp_action: "finite (aexp_action α)"
⟨*proof*⟩

fun aexp_edge :: "('n,'v action) edge ⇒ 'v arith set" **where**
"aexp_edge (q1, α, q2) = aexp_action α"

lemma finite_aexp_edge: "finite (aexp_edge (q1, α, q2))"
⟨*proof*⟩

fun aexp_pg :: "('n,'v action) program_graph ⇒ 'v arith set" **where**
"aexp_pg pg = ⋃(aexp_edge ‘ (fst pg))"

definition aexp_edge_list :: "('n,'v action) edge list ⇒ 'v arith ⇒ bool" **where**
"aexp_edge_list π a = (∃π1 π2 e. π = π1 @ [e] @ π2 ∧ a ∈ aexp_edge e ∧ (∀e' ∈ set ([e] @ π2). fv_arith a ∩ def_edge e' = {}))"

definition aexp_path :: "'n list × 'v action list ⇒ 'v arith set" **where**
"aexp_path π = {a. aexp_edge_list (transition_list π) a}"

locale analysis_AE = finite_program_graph pg
for pg :: "('n::finite,'v::finite action) program_graph"
begin

interpretation LTS edges ⟨*proof*⟩

definition analysis_dom_AE :: "'v arith set" **where**
"analysis_dom_AE = aexp_pg pg"

lemma finite_analysis_dom_AE: "finite analysis_dom_AE"
⟨*proof*⟩

fun kill_set_AE :: "('n,'v action) edge ⇒ 'v arith set" **where**
"kill_set_AE (q_o, x ::= a, q_s) = {a'. x ∈ fv_arith a'}"
| "kill_set_AE (q_o, Bool b, q_s) = {}"
| "kill_set_AE (v, Skip, vc) = {}"

fun gen_set_AE :: "('n,'v action) edge ⇒ 'v arith set" **where**
"gen_set_AE (q_o, x ::= a, q_s) = {a'. a' ∈ ae_arith a ∧ x ∉ fv_arith a'}"
| "gen_set_AE (q_o, Bool b, q_s) = ae_boolean b"
| "gen_set_AE (v, Skip, vc) = {}"

definition d_init_AE :: "'v arith set" **where**
"d_init_AE = {}"

interpretation fw_must: analysis_BV_forward_must pg analysis_dom_AE kill_set_AE gen_set_AE d_init_AE
⟨*proof*⟩

lemma aexp_edge_list_S_hat_edge_list:
assumes "a ∈ aexp_edge (q, α, q')"
assumes "fv_arith a ∩ def_edge (q, α, q') = {}"
shows "a ∈ fw_must.S_hat (q, α, q') R"
⟨*proof*⟩

lemma empty_inter_fv_arith_def_edge:
assumes "aexp_edge_list (π @ [e]) a"
shows "fv_arith a ∩ def_edge e = {}"
⟨*proof*⟩

lemma aexp_edge_list_append_singleton:
assumes "aexp_edge_list (π @ [e]) a"
shows "aexp_edge_list π a ∨ a ∈ aexp_edge e"
⟨*proof*⟩

lemma gen_set_AE_subset_aexp_edge:
assumes "a ∈ gen_set_AE e"
shows "a ∈ aexp_edge e"
⟨*proof*⟩

lemma empty_inter_fv_arith_def_edge':
assumes "a ∈ gen_set_AE e"
shows "fv_arith a ∩ def_edge e = {}"
⟨*proof*⟩

lemma empty_inter_fv_arith_def_edge'':
assumes "a ∉ kill_set_AE e"
shows "fv_arith a ∩ def_edge e = {}"
⟨*proof*⟩

lemma S_hat_edge_list_aexp_edge_list:
assumes "a ∈ fw_must.S_hat_edge_list π d_init_AE"
shows "aexp_edge_list π a"
⟨*proof*⟩

lemma not_kill_set_AE_iff_fv_arith_def_edge_disjoint:
"fv_arith a ∩ def_edge e = {} ⟷ a ∉ kill_set_AE e"
⟨*proof*⟩

lemma gen_set_AE_AE_iff_fv_arith_def_edge_disjoint_and_aexp_edge:
"a ∈ aexp_edge e ∧ fv_arith a ∩ def_edge e = {} ⟷ a ∈ gen_set_AE e"
⟨*proof*⟩

lemma aexp_edge_list_S_hat_edge_list':
assumes "aexp_edge_list π a"
shows "a ∈ fw_must.S_hat_edge_list π d_init_AE"
⟨*proof*⟩

lemma aexp_edge_list_S_hat_edge_list_iff:
"aexp_edge_list π a ⟷ a ∈ fw_must.S_hat_edge_list π d_init_AE"
⟨*proof*⟩

lemma aexp_path_S_hat_path_iff:
"a ∈ aexp_path π ⟷ a ∈ fw_must.S_hat_path π d_init_AE"
⟨*proof*⟩

definition summarizes_AE :: "(pred, ('n, 'a, 'v arith) cst) pred_val ⇒ bool" **where**
"summarizes_AE ϱ ⟷

```
    (∀q d.
        ϱ ⊨_lh must⟨[q, d]⟩. ⟶
         (∀π. π ∈ path_with_word_from_to start (the_Node_id q) ⟶ (the_Elem_id d) ∈ aexp_path π))"

theorem AE_sound:
  assumes "ϱ ⊨_lst (fw_must.ana_pg_fw_must) s_BV"
  shows "summarizes_AE ϱ"
⟨proof⟩

end

end
theory Very_Busy_Expressions imports Available_Expressions begin
```

— We encode the Very Busy Expressions analysis into Datalog. First we define the analysis, then we encode it into Datalog using our Bit-Vector Framework locale. We also prove the encoding correct.

15 Very Busy Expressions

```
definition vbexp_edge_list :: "('n,'v action) edge list ⇒ 'v arith ⇒ bool" where
  "vbexp_edge_list π a = (∃π1 π2 e. π = π1 @ [e] @ π2 ∧ a ∈ aexp_edge e ∧ (∀e' ∈ set π1. fv_arith
a ∩ def_edge e' = {}))"

definition vbexp_path :: "'n list × 'v action list ⇒ 'v arith set" where
  "vbexp_path π = {a. vbexp_edge_list (LTS.transition_list π) a}"

locale analysis_VB = finite_program_graph pg
  for pg :: "('n::finite,'v::finite action) program_graph"
begin

interpretation LTS edges ⟨proof⟩

definition analysis_dom_VB :: "'v arith set" where
  "analysis_dom_VB = aexp_pg pg"

lemma finite_analysis_dom_VB: "finite analysis_dom_VB"
⟨proof⟩

fun kill_set_VB :: "('n,'v action) edge ⇒ 'v arith set" where
  "kill_set_VB (q_o, x ::= a, q_s) = {a'. x ∈ fv_arith a'}"
| "kill_set_VB (q_o, Bool b, q_s) = {}"
| "kill_set_VB (v, Skip, vc) = {}"

fun gen_set_VB :: "('n,'v action) edge ⇒ 'v arith set" where
  "gen_set_VB (q_o, x ::= a, q_s) = ae_arith a"
| "gen_set_VB (q_o, Bool b, q_s) = ae_boolean b"
| "gen_set_VB (v, Skip, vc) = {}"

definition d_init_VB :: "'v arith set" where
  "d_init_VB = {}"

interpretation bw_must: analysis_BV_backward_must pg analysis_dom_VB kill_set_VB gen_set_VB d_init_VB
  ⟨proof⟩

lemma aexp_edge_list_S_hat_edge_list:
  assumes "a ∈ aexp_edge (q, α, q')"
  assumes "fv_arith a ∩ def_edge (q, α, q') = {}"
  shows "a ∈ bw_must.S_hat (q, α, q') R"
  ⟨proof⟩

lemma empty_inter_fv_arith_def_edge:
  assumes "vbexp_edge_list (e # π) a"
  shows "fv_arith a ∩ def_edge e = {} ∨ a ∈ aexp_edge e"
```

⟨*proof*⟩

lemma vbexp_edge_list_Cons:
  assumes "vbexp_edge_list (e # π) a"
  shows "vbexp_edge_list π a ∨ a ∈ aexp_edge e"
⟨*proof*⟩

lemma gen_set_VB_is_aexp_edge:
  "a ∈ gen_set_VB e ⟷ a ∈ aexp_edge e"
⟨*proof*⟩

lemma empty_inter_fv_arith_def_edge'':
  "a ∉ kill_set_VB e ⟷ fv_arith a ∩ def_edge e = {}"
⟨*proof*⟩

lemma S_hat_edge_list_aexp_edge_list:
  assumes "a ∈ bw_must.S_hat_edge_list π d_init_VB"
  shows "vbexp_edge_list π a"
  ⟨*proof*⟩

lemma aexp_edge_list_S_hat_edge_list':
  assumes "vbexp_edge_list π a"
  shows "a ∈ bw_must.S_hat_edge_list π d_init_VB"
  ⟨*proof*⟩

lemma vbexp_edge_list_S_hat_edge_list_iff:
  "vbexp_edge_list π a ⟷ a ∈ bw_must.S_hat_edge_list π d_init_VB"
  ⟨*proof*⟩

lemma vbexp_path_S_hat_path_iff:
  "a ∈ vbexp_path π ⟷ a ∈ bw_must.S_hat_path π d_init_VB"
  ⟨*proof*⟩

definition summarizes_VB **where**
  "summarizes_VB ϱ ⟷
    (∀q d.
        ϱ ⊨$_{lh}$ must⟨[q, d]⟩. ⟶
         (∀π. π ∈ path_with_word_from_to (the_Node$_{id}$ q) end ⟶ the_Elem$_{id}$ d ∈ vbexp_path π))"

theorem VB_sound:
  assumes "ϱ ⊨$_{lst}$ (bw_must.ana_pg_bw_must) s_BV"
  shows "summarizes_VB ϱ"
⟨*proof*⟩

end

end
theory Reachable_Nodes **imports** Bit_Vector_Framework **begin**

— We define an analysis for collecting the reachable nodes in a program graph. First we define it analysis, and then we encode it into Datalog using our Bit-Vector Framework locale. We also prove the encoding correct.

16 Reachable Nodes

fun nodes_on_edge :: "('n,'v) edge ⇒ 'n set" **where**
  "nodes_on_edge (q1, α, q2) = {q1, q2}"

definition node_on_edge_list :: "('n,'v) edge list ⇒ 'n ⇒ bool" **where**
  "node_on_edge_list π q = (∃π1 π2 e. π = π1 @ [e] @ π2 ∧ q ∈ nodes_on_edge e)"

definition nodes_on_path :: "'n list × 'v action list ⇒ 'n set" **where**
  "nodes_on_path π = {q. node_on_edge_list (LTS.transition_list π) q}"

locale analysis_RN = finite_program_graph pg
  for pg :: "('n::finite,'v::finite action) program_graph"
begin

interpretation LTS edges ⟨*proof*⟩

definition analysis_dom_RN :: "'n set" **where**
  "analysis_dom_RN = UNIV"

fun kill_set_RN :: "('n,'v action) edge ⇒ 'n set" **where**
  "kill_set_RN (q_o, α, q_s) = {}"

fun gen_set_RN :: "('n,'v action) edge ⇒ 'n set" **where**
  "gen_set_RN (q_o, α, q_s) = {q_o}"

definition d_init_RN :: "'n set" **where**
  "d_init_RN = {end}"

interpretation bw_may: analysis_BV_backward_may pg analysis_dom_RN kill_set_RN gen_set_RN d_init_RN
  ⟨*proof*⟩

lemma node_on_edge_list_S_hat_edge_list:
  assumes "ts ∈ transition_list_path"
  assumes "trans_tl (last ts) = end"
  assumes "node_on_edge_list ts q"
  shows "q ∈ bw_may.S_hat_edge_list ts d_init_RN"
  ⟨*proof*⟩

lemma S_hat_edge_list_node_on_edge_list:
  assumes "π ≠ []"
  assumes "trans_tl (last π) = end"
  assumes "q ∈ bw_may.S_hat_edge_list π d_init_RN"
  shows "node_on_edge_list π q"
  ⟨*proof*⟩

lemma node_on_edge_list_UNIV_S_hat_edge_list:
  assumes "π ∈ transition_list_path"
  assumes "π ≠ []"
  assumes "trans_tl (last π) = end"
  shows "{q. node_on_edge_list π q} = bw_may.S_hat_edge_list π d_init_RN"
  ⟨*proof*⟩

lemma nodes_singleton_if_path_with_word_empty':
  assumes "(ss, w) ∈ path_with_word"
  assumes "w = []"
  shows "∃l. ss = [l]"
⟨*proof*⟩

lemma nodes_singleton_if_path_with_word_empty:
  assumes "(ss, []) ∈ path_with_word"
  shows "∃l. ss = [l]"
  ⟨*proof*⟩

lemma path_with_word_append_last:
  assumes "(ss,w) ∈ path_with_word"
  assumes "w ≠ []"
  shows "last (transition_list (s # ss, l # w)) = last (transition_list (ss, w))"
  ⟨*proof*⟩

lemma last_trans_tl:
  assumes "(ss,w) ∈ path_with_word"
  assumes "w ≠ []"

shows "last ss = trans_tl (last (LTS.transition_list (ss,w)))"
⟨*proof*⟩

lemma nodes_tail_empty_if_path_with_word_empty:
assumes "(ss,w) ∈ path_with_word"
assumes "w ≠ []"
shows "transition_list (ss,w) ≠ []"
⟨*proof*⟩

lemma empty_transition_list_if_empty_word:
assumes "π ∈ path_with_word"
assumes "snd π ≠ []"
shows "transition_list π ≠ []"
⟨*proof*⟩

lemma two_nodes_if_nonempty_word:
assumes "(ss, w) ∈ path_with_word"
assumes "w ≠ []"
shows "∃s' s'' ss'. ss = s' # s'' # ss'"
⟨*proof*⟩

lemma path_with_word_empty_word:
assumes "(ss', w) ∈ path_with_word"
assumes "ss' = s' # ss"
assumes "w = []"
shows "ss = []"
⟨*proof*⟩

lemma transition_list_path_if_path_with_word:
assumes "(ss,w) ∈ path_with_word"
assumes "w ≠ []"
shows "transition_list (ss,w) ∈ transition_list_path"
⟨*proof*⟩

lemma nodes_on_path_S_hat_path:
assumes "π ∈ path_with_word"
assumes "snd π ≠ []"
assumes "last (fst π) = end"
shows "nodes_on_path π = bw_may.S_hat_path π d_init_RN"
⟨*proof*⟩

definition summarizes_RN **where**
"summarizes_RN ϱ ⟷ (∀π d. π ∈ path_with_word_to end ⟶ d ∈ nodes_on_path π ⟶
ϱ ⊨$_{lh}$ may⟨[Cst$_N$ (start_of π), Cst$_E$ d]⟩.)"

theorem RN_sound:
assumes "ϱ ⊨$_{lst}$ bw_may.ana_pg_bw_may s_BV"
shows "summarizes_RN ϱ"
⟨*proof*⟩

end

end

References

[BCD17] Véronique Benzaken, Evelyne Contejean, and Stefania Dumbrava. Certifying standard and stratified Datalog inference engines in SSReflect. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 171–188. Springer, 2017.

[BDA18] Angela Bonifati, Stefania Dumbrava, and Emilio Jesús Gallego Arias. Certified graph view maintenance with Regular Datalog. *Theory Pract. Log. Program.*, 18(3-4):372–389, 2018.

[Bre10] Joachim Breitner. Shivers' control flow analysis. *Archive of Formal Proofs*, November 2010. https://isa-afp.org/entries/Shivers-CFA.html, Formal proof development.

[Bre15] Joachim Breitner. The safety of Call Arity. *Archive of Formal Proofs*, February 2015. https://isa-afp.org/entries/Call_Arity.html, Formal proof development.

[Bre16] Joachim Breitner. *Lazy Evaluation: From natural semantics to a machine-checked compiler transformation*. PhD thesis, Karlsruhe Institute of Technology, 2016.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252, 1977.

[CP10] David Cachera and David Pichardie. A certified denotational abstract interpreter. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 9–24. Springer, 2010.

[Dum16] Stefania-Gabriela Dumbrava. *A Coq Formalization of Relational and Deductive Databases -and Mechanizations of Datalog*. PhD thesis, University of Paris-Saclay, France, 2016.

[Imm11] Fabian Immler. RIPEMD-160. *Archive of Formal Proofs*, January 2011. https://isa-afp.org/entries/RIPEMD-160-SPARK.html, Formal proof development.

[Jia21] Nan Jiang. A data flow analysis algorithm for computing dominators. *Archive of Formal Proofs*, September 2021. https://isa-afp.org/entries/Dominance_CHK.html, Formal proof development.

[JLB+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL 2015)*, pages 247–259, 2015.

[KKB13] Jael Kriener, Andy King, and Sandrine Blazy. Proofs you can believe in: proving equivalences between Prolog semantics in Coq. In Ricardo Peña and Tom Schrijvers, editors, *15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, Madrid, Spain, September 16-18, 2013*, pages 37–48. ACM, 2013.

[LM08] Peter Lammich and Markus Müller-Olm. Conflict analysis of programs with procedures, dynamic thread creation, and monitors. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*, volume 5079 of *Lecture Notes in Computer Science*, pages 205–220. Springer, 2008.

[LMO07] Peter Lammich and Markus Müller-Olm. Formalization of conflict analysis of programs with procedures, thread creation, and monitors. *Archive of Formal Proofs*, December 2007. https://isa-afp.org/entries/Program-Conflict-Analysis.html, Formal proof development.

[Nip12] Tobias Nipkow. Abstract interpretation of annotated commands. In *Proceedings of the Third International Conference on Interactive Theorem Proving (ITP 2012)*, pages 116–132, 2012.

[NK14] Tobias Nipkow and Gerwin Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014.

[NN20] Flemming Nielson and Hanne Riis Nielson. Program analysis (an appetizer). *CoRR*, abs/2012.10086, 2020.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[Prz88] Teodor C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.

[SHN24] Anders Schlichtkrull, René Rydhof Hansen, and Flemming Nielson. Isabelle-verified correctness of datalog programs for program analysis. In Jiman Hong and Juw Won Park, editors, *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing, SAC 2024, Avila, Spain, April 8-12, 2024*, pages 1731–1734. ACM, 2024.

[SSST22] Anders Schlichtkrull, Morten Konggaard Schou, Jirí Srba, and Dmitriy Traytel. Differential testing of pushdown reachability with a formally verified oracle. In Alberto Griggio and Neha Rungta, editors, *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, pages 369–379. IEEE, 2022.

[SSST23] Anders Schlichtkrull, Morten Konggaard Schou, Jií Srba, and Dmitriy Traytel. Labeled transition systems. *Archive of Formal Proofs*, October 2023. https://isa-afp.org/entries/Labeled_Transition_Systems.html, Formal proof development.

[TGMK25] Johannes Tantow, Lukas Gerlach, Stephan Mennicke, and Markus Krötzsch. Verifying Datalog reasoning with Lean. In *Proceedings of the 16th International Conference on Interactive Theorem Proving (ITP 2025)*, 2025.

[Was08] Daniel Wasserrab. Towards certified slicing. *Archive of Formal Proofs*, September 2008. https://isa-afp.org/entries/Slicing.html, Formal proof development.

[Was09] Daniel Wasserrab. Backing up slicing: Verifying the interprocedural two-phase Horwitz-Reps-Binkley slicer. *Archive of Formal Proofs*, November 2009. https://isa-afp.org/entries/HRB-Slicing.html, Formal proof development.

[Whi06] Nathan Whitehead. A certified distributed security logic for authorizing code. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs, International Workshop, TYPES 2006, Nottingham, UK, April 18-21, 2006, Revised Selected Papers*, volume 4502 of *Lecture Notes in Computer Science*, pages 253–268. Springer, 2006.

[WL08] Daniel Wasserrab and Andreas Lochbihler. Formalizing a framework for dynamic slicing of program dependence graphs in isabelle/hol. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2008.

[WLS09] Daniel Wasserrab, Denis Lohner, and Gregor Snelting. On PDG-based noninterference and its modular proof. In Stephen Chong and David A. Naumann, editors, *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, pages 31–44. ACM, 2009.